


Reproducible Floating-Point Aggregation in RDBMSs

Extended version

Working Paper

Author(s):

Müller, Ingo ; Arteaga, Andrea; Hoefler, Torsten; Alonso, Gustavo

Publication date:

2018-02-27

Permanent link:

<https://doi.org/10.3929/ethz-b-000409592>

Rights / license:

[Creative Commons Attribution 4.0 International](#)

Originally published in:

arXiv

Reproducible Floating-Point Aggregation in RDBMSs

[Extended Version]

Ingo Müller^{1*}

Andrea Arteaga²

Torsten Hoefler¹

Gustavo Alonso¹

¹*Systems Group, Dept. of Computer Science, ETH Zurich*
{ingo.mueller, torsten.hoefler, alonso}@inf.ethz.ch

²*Federal Institute of Meteorology and Climatology MeteoSwiss*
andrea.arteaaga@meteoswiss.ch

Abstract—Industry-grade database systems are expected to produce the same result if the same query is repeatedly run on the same input. However, the numerous sources of non-determinism in modern systems make reproducible results difficult to achieve. This is particularly true if floating-point numbers are involved, where the order of the operations affects the final result.

As part of a larger effort to extend database engines with data representations more suitable for machine learning and scientific applications, in this paper we explore the problem of making relational GROUPBY over floating-point formats *bit-reproducible*, i.e., ensuring any execution of the operator produces the same result up to every single bit. To that aim, we first propose a numeric data type that can be used as drop-in replacement for other number formats and is—unlike standard floating-point formats—associative. We use this data type to make state-of-the-art GROUPBY operators reproducible, but this approach incurs a slowdown between $4 \times$ and $12 \times$ compared to the same operator using conventional database number formats. We thus explore how to modify existing GROUPBY algorithms to make them bit-reproducible and efficient. By using vectorized summation on batches and carefully balancing batch size, cache footprint, and preprocessing costs, we are able to reduce the slowdown due to reproducibility to a factor between $1.9 \times$ and $2.4 \times$ of aggregation in isolation and to a mere 2.7% of end-to-end query performance even on aggregation-intensive queries in MonetDB. We thereby provide a solid basis for supporting more reproducible operations directly in relational engines.

This document is an extended version of an article currently in print for the proceedings of ICDE'18 with the same title and by the same authors. The main additions are more implementation details and experiments.

I. INTRODUCTION

The continued progress of all areas of computer science has led to digitization and automation of many everyday processes. In particular, algorithms are responsible for making decisions in an increasing number of commonplace situations [20]. Consequently—and understandably—, society has started to demand accountability from the algorithms by which it is affected. For example, the General Data Protection Regulation of the European Union [19] has recently given the “right to explanation” to individuals affected by automated decision-making. Similarly, the ACM published a *Statement on Algorithmic Transparency and Accountability* [1] including explainability as a principle that algorithmic decision-making

*Part of this work was carried out while this author was working part-time at Oracle Labs, Zurich.

```
CREATE TABLE R (i int, f float);
INSERT INTO R VALUES (1, 2.5e-16);
INSERT INTO R VALUES (2, 0.9999999999999999);
INSERT INTO R VALUES (3, 2.5e-16);
SELECT SUM(f) FROM R; -- Returns 0.9999999999999999
UPDATE R SET i = i + 1 WHERE i = 2;
-- 'f' is unchanged, but rows are physically reordered
SELECT SUM(f) FROM R; -- Returns 1.0!
```

Algorithm 1: Example of non-reproducible SQL query.

should follow, a problem studied for example in the context of computational journalism [15, 16].

In this paper we take steps towards improving explainability of today’s data processing systems, namely the reproducibility of algorithms based on floating-point arithmetic. This problem was brought to our attention by several of our industry partners. In addition to more classical use-cases like debugging, testing, certification, and redundant computations, where reproducibility can be helpful or necessary [8, 10], they observe that many users, in particular non-experts, are confused by non-reproducible or, in general, non-predictable behavior.

Today’s data management systems often become non-reproducible if floating-point arithmetic is used. The problem with floating-point arithmetic is that, unlike arithmetic on real numbers, it is not associative, so the order in which operations are carried out may change their outcome [21]. The order of operations, in turn, may be affected by a large list of mechanisms: For example, concurrent execution of multiple threads may be non-deterministic, the number of processing elements may influence how the work is split into sub-tasks, and the data storage layer may physically reorder data for a number of reasons. As soon as floating-point numbers are involved, most of today’s systems do not follow the principle of *data independence*, which demands that changes on the physical level shall not have an impact on the result of queries.

Algorithm 1 illustrates how the order of records in the storage layer may affect query results in a subtle and potentially surprising way. The situation shown was produced on a fresh installation of PostgreSQL 9.5.1. The same query summing up three floating-point numbers returns two different results before and after the update of an unrelated attribute. Since, internally, the update is implemented as the creation of a new record and the masking of the old one, the physical order is different

in the two queries, which, consequently, yields two different results (differing on all digits of the decimal representation).

One may be tempted to brush away the problem of non-reproducibility with the argument that the underlying rounding errors are rather small and can, hence, be ignored. However, these small errors may still lead to very different outcomes for individual records, which are hard to explain to the affected individual. For instance, the GROUPBY query of Algorithm 1 extended with a HAVING SUM(f) ≥ 1 clause could end up returning specific records in some runs but not in others.

Such misclassifications can affect applications in obvious ways: We ran PageRank on different permutations of a small web graph with 900 k pages.¹ We observed that, from one run to the next, the ranks of about 10-20 pages would be *different enough to swap ranks with another page*.

In this paper we show how to make GROUPBY aggregation using SUM reproducible. This essentially solves the problem for SQL: With a reproducible aggregate function for floating-point SUM, *all* aggregate functions in SQL can be made reproducible as well, including non-standard ones such as VARIANCE, STDDEV, and some statistical functions, all of which can be computed using SUM.² Furthermore, many projections are intrinsically reproducible³ and window functions can either be solved with our approach, define an execution order, or they are not reproducible even with integers.⁴ Finally, GROUPBY aggregation is not only used in relational database systems, but in virtually every data processing system (possibly under a different name including REDUCE or REDUCEBYKEY), to which our results apply as well.

We start with proposing a format for floating-point numbers that is—unlike formats typically supported by hardware—associative. The format can be implemented in software and builds on techniques from high-performance computing (HPC) [3, 13, 14]. The key to this approach is to anticipate rounding errors by subtracting lower-order terms from each value before it is added to the aggregate of its group.

While this makes it possible to make any algorithm on floating-point numbers bit-reproducible with little to no modification, it comes at a high price: We show that state-of-the-art GROUPBY operators become about $4\times$ and $12\times$ slower using this approach, depending on the desired precision. The challenge is, hence, to keep the overhead of the additional

¹<https://snap.stanford.edu/data/web-Google.html>

²Aggregate functions such as MIN, MEDIAN, COUNT, RANK, PERCENTILE, and similar, but also functions such as LISTAGG and COLLECT, do not need floating-point arithmetic. The remaining functions offered by the Oracle database (see <https://docs.oracle.com/database/121/SQLRF/functions003.htm>) can be computed with SUM.

³If arithmetic expressions are computed in their entirety only once all operands are available, then executing them always in the same order is trivial. However, if expressions are broken up and partially pushed through joins, then their execution order may depend on the join order, which may change even if the logical input has not changed. However, we believe that this should be solved on the level of the optimizer, which is out of the scope of this paper.

⁴Window clauses without sliding frame can be executed as aggregations with GroupBy. Window clauses with ORDERBY clause have a definite order and are therefore intrinsically reproducible and without ORDERBY clause, the result may change even with integers.

calculations at an acceptable level. However, the tuning techniques used in HPC work for the sum of a single vector, while in a SQL GROUPBY, there is potentially a very large number of sums involved. This is not compatible with known data processing techniques, which usually aggregate input tuples as early as possible instead of physically “grouping” them. State-of-the-art aggregation algorithms used with our data type, hence, switch between the summation of different groups *for every input tuple*, which explains the high overhead.

To remedy this problem, we design a novel GROUPBY algorithm based on a concept we call *summation buffer*. The main idea is to buffer input values for each group and delay their aggregation until it can be done efficiently for the whole buffer. As we need a summation buffer for every group, the number of groups that we can process efficiently at the same time is limited by how many buffers we can keep in cache. We thus tune the summation routine to small buffer sizes and use highly-tuned partitioning routines as preprocessing. This reduces the slowdown of aggregations due to reproducibility to a factor between $1.9\times$ and $2.4\times$ over non-reproducible aggregation on built-in floating-point numbers, depending on the number of groups in the input. Integration into a real system, MonetDB [9], shows that we can bring the overhead of end-to-end query performance down to as little as 2.7%. Since our implementation hides almost all computations behind memory accesses, we can even increase accuracy with minimal additional cost and, hence, as a side effect provide higher accuracy than IEEE numbers at essentially the same price, which is crucial in many scientific applications.

To summarize, the paper makes the following contributions:

- We propose a highly tuned algorithm for reproducible summation of floating-point numbers using SIMD instructions (Section III).
- We show how state-of-the-art algorithms for aggregation with GROUPBY can be made bit-reproducible and more accurate with relatively little effort if compromises in performance can be made (Section IV).
- We design a novel grouping algorithm that improves upon this approach, which reduces the slowdown of reproducibility to a $1.9\times$ and $2.4\times$ (Section V).
- We show the trade-offs offered by the different algorithms in extensive experiments and quantify their impact on end-to-end query performance in a real system (Section VI).

II. PROBLEM DEFINITION

We start by illustrating the cause of non-reproducibility of floating-point summation and by discussing potential solutions for bit-reproducibility, which, unfortunately, either do not actually solve the problem or have prohibitive costs.

A. Reproducible Floating-Point Aggregation with GROUPBY

For the purpose of this paper, we define aggregation with GROUPBY as the operation that turns a sequence of $\langle key, value \rangle$ pairs into the $\langle key, aggregate \rangle$ pairs where each key of the input occurs exactly once in the output and the aggregate stored with a key is equal to the sum of all values with that

key in the input. We say that it runs on floating-point numbers if the *value* fields of the input pairs are floating-point numbers. An aggregation algorithm is bit-reproducible, or reproducible for short, if the *aggregate* of each group has exactly the same bit pattern for any execution.

B. Floating-Point Numbers and Associativity

Floating-point values are numbers of the form $x = M \cdot 2^E$, where $M \in [1, 2)$ is called *mantissa* and $E \in \{E_{min}, E_{max}\}$ is the *exponent*. As the number of relevant bits m in the mantissa as well as the exponent are finite, only a finite subset of real numbers can be represented exactly. Hence, a rounding function rd is required in order to map real numbers to representable floating-point values. This includes the results of arithmetic expressions, which may not be representable even if the operands are. For example, the floating-point sum of two floating-point numbers a and b is defined as $a \oplus b = rd(a + b)$.

To understand why this can be a problem, consider the numbers $a = b = 1.01_2 \cdot 2^0$ and $c = 1.11_2 \cdot 2^1$ in a toy format for floating-point numbers with a mantissa of $m = 2$ (given as binary number) and truncation for rd . To compute the sum of the three numbers, we can compute $(a \oplus b) \oplus c = rd(rd(a + b) + c)$. Since $rd(a + b) = rd(1.010_2 \cdot 2^1) = 1.01_2 \cdot 2^1$ and $rd(1.01_2 \cdot 2^1 + c) = rd(1.100_2 \cdot 2^2) = 1.10_2 \cdot 2^2$, no rounding errors occur and the sum is accurate. However, we can compute the sum as well as $a \oplus (b \oplus c) = rd(a + rd(b + c))$. In this case $rd(b + c) = rd(1.0011_2 \cdot 2^2) = 1.00_2 \cdot 2^2$ and $rd(a + 1.00_2 \cdot 2^2) = rd(1.0101_2 \cdot 2^2) = 1.01_2 \cdot 2^2$, so rounding errors occur in both operations (typeset in bold). Note that the *sum* of the rounding errors is $1.00_2 \cdot 2^0$, which could be added to $a \oplus (b \oplus c)$ without rounding error.

Rounding errors are larger if the exponents of the two summands are different. Therefore, if we compute the sum of many numbers, the rounding error incurred during the addition of a particular input value depends on the current value of the accumulator, which depends on the order of execution. Furthermore, even though each error is small, their sum may be big enough to change the final result.

The problem also occurs in the most common floating-point formats, which are the ones defined by the IEEE-754 standard [39] (even if the absolute error is obviously smaller due to the higher precision than our toy format) and which we use throughout this paper.

C. Non-Solutions for Reproducibility

We now discuss a number of naive approaches for making aggregation with GROUPBY reproducible, but which do not provide satisfactory solutions to the problem.

Higher precision. Using a truncated or rounded result produced by operations with a higher floating-point precision (i.e., using doubles instead of floats) is not sufficient, as it does not make it more reproducible: Even tiny rounding errors can make significant bits flip (such as from 0.999999... to 1).

Deterministic order of operations. It is possible to make the order of the operations deterministic. For linear algebra, the cuBLAS library [27] and the Intel Math Kernel Library [32]

follow this approach. However, this does not solve the entire problem for database systems, which aim for *data independence* as discussed above. In addition to the example given in the introduction, the physical order of the input may also change due to compression, data placement on distributed machines, backup and restore operations, and other mechanisms, which in turn changes the order of operations. The only way to make the order of the operations deterministic is thus to use a static and deterministic schedule *and* to sort the input,⁵ which may be more than an order of magnitude slower [4, Figure 5.8] than state-of-the-art AGGREGATION algorithms based on hashing.

Fixed-point arithmetic. In traditional workloads, it is often possible to use fixed-point arithmetic for fractional numbers (also called *binary scaling* or *decimal-scaled binaries* if the scale has base ten). This is the case if all input numbers are integer multiple of some common denominator and come from similar orders of magnitude, such as all values in a *salary* field are multiples of 1¢ and range between some thousand and some million dollars. Operations can then be executed using integer operations internally, which are cheap to execute and reproducible most of the time.⁶ However, in many modern data processing applications, these assumptions do not apply: values from some domains, such as measurements or scientific data, cannot be expressed as multiple of some smallest unit and the values of different orders of magnitude such as those handled in machine learning and other scientific applications require a floating-point representation.

Arbitrary-precision operations. It is possible to push the limits of fixed-point arithmetic by using high-precision or even arbitrary-precision operations in software. Examples implementing this approach include the GNU MPFR Library [35], the BigDecimal class of Java [29], and numeric data types offered by some database systems (such as PostgreSQL). However, this not only requires many hardware instructions for each arithmetic operation, but also variable-width storage, which is much more difficult to handle than the fixed-width built-in types. Similar arguments apply to *unums* [22].

III. REPRODUCIBLE ACCURATE SUM

In this section, we explain an algorithm that solves the problem of reproducible floating-point summation where all inputs are summed up to produce a *single* number (i.e., aggregation *without* grouping). For the sake of exposition, we develop the algorithm in three iterations of increasing completeness. In a fourth iteration, we show how to speed up this algorithm using vector instructions. Table I summarizes the variables and function names we use.

⁵Provably so: Visiting a set of elements in a fixed order solves the PROXIMATE NEIGHBOR problem, which is as hard as SORTING [23].

⁶Summing integers may experience overflows, which can lead to non-reproducible results if the values have mixed signs and depending on how they are handled. If overflows are to be prevented in software, this may incur a slowdown as high as $3 \times$ as well [17].

Name	Meaning
n	Number of input values
L	Number of levels
V	Size of vector register
W	Bit width of error-free transformation
NB	Block size between carry-bit operations
m	Size of the mantissa
f	Exponent of the first transformation level
b_i	Input value
$q_i^{(l)}$	Contribution of b_i at level l
$r_i^{(l)}$	Remainder of b_i at level l
$S^{(l)}$	Running sum at level l
$C^{(l)}$	Carry-bit count at level l
$Q^{(l)}$	Sum of the contributions at level l
$\text{ufp}(x)$	Unit in the first place
$\text{ulp}(x)$	Unit in the last place
$\text{rd}(x)$	Rounding function
$x \oplus y$	$\equiv \text{rd}(x + y)$, floating-point sum of x and y
$x \ominus y$	$\equiv \text{rd}(x - y)$, floating-point subtraction of x and y

Table I: Summary of the parameters, variables, and functions used in Section III.

A. Definitions

Two numbers related to any floating-point number x are of importance in this section: $\text{ufp}(x)$ and $\text{ulp}(x)$. They were first defined by Goldberg [21]. $\text{ufp}(x)$ designates the *unit in the first place*, i.e., the numeric value of the first bit in the mantissa. If $x = M \cdot 2^E$ as above, then $\text{ufp}(x) = 2^E$. All floating-point numbers with the same exponent as x have an absolute value in $[\text{ufp}(x), 2 \cdot \text{ufp}(x))$. Similarly, $\text{ulp}(x)$ designates the *unit in the last place*, i.e., the value of the last bit in the mantissa. For $x = M \cdot 2^E$, $\text{ulp}(x) = 2^{E-m}$. This value represents the difference between x and its closest representable values.

B. Error-Free Transformation

Our reproducible summation algorithm is based on the observation that the floating-point sum of two numbers a and b can be performed *exactly* if the one with the smaller absolute value has sufficiently many zeroes at the end of its mantissa. To understand when this is the case, let us define a and b as integer multiples of the same power of two: $a := \alpha_a \cdot 2^p$, $b := \alpha_b \cdot 2^p$, where α_a, α_b , and p are integer numbers, and $|a| > |b|$, WLOG. If the values $|\alpha_a|, |\alpha_b|$, and $|\alpha_a + \alpha_b|$ do not exceed 2^m , then a, b , and $a + b$ can be represented in the floating-point format and, consequently, the floating-point sum is exact: $a \oplus b = a + b$. In other words, the sum is exact if these three numbers can have a (possibly denormalized) floating-point representation with the same exponent and without losing information.

As an example, the values $a := 26.046875$ and $b := 2.8125$ can be represented exactly with an 11-bit mantissa (which corresponds to IEEE half-precision, where $m = 10$). Also, they can be represented as $\alpha_a \cdot 2^p$ and $\alpha_b \cdot 2^p$, with $\alpha_a = 1667$, $\alpha_b = 180$, and $p = -6$ and their sum is $a \oplus b = a + b = (\alpha_a + \alpha_b) \cdot 2^p = 28.859375$. It can be computed exactly with this format because said conditions are met.

Let us now consider any representable a and b with $|a| > |b|$. One can split b as $b = q + r$ so that q is an integer multiple of

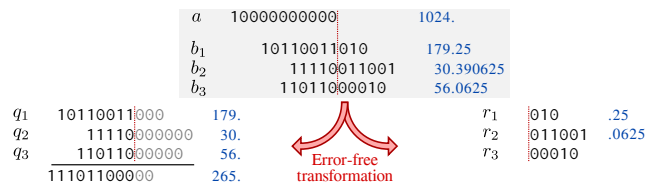


Figure 1: Error-free transformation.

$\text{ulp}(a)$, namely $q := (a \oplus b) \ominus a$ and $r := b \ominus q$. The sum $a \oplus q$ can be computed exactly because the three aforesaid conditions are met between a and q , with $2^p = \text{ulp}(a)$. Also, b can be recovered exactly through $q + r = q \oplus b$. This procedure, named *error-free transformation*, was defined by Ogita et al. [28].

For instance, $a = 1.010_2 \cdot 2^0$, $b = 1.101_2 \cdot 2^{-2}$. q is computed as $(a \oplus b) \ominus b = 1.101_2 \cdot 2^0$, while $r = b \ominus q = 1_2 \cdot 2^{-5}$.

We can now perform an order-independent summation of a sequence of values b_1, b_2, b_3 by applying the same error-free transformation to all values b_i . Figure 1 illustrates the idea. The transformation produces $q_i := (a \oplus b_i) \ominus a$, $r_i := b_i \ominus q_i$. We call the values q_i and r_i *contributions* and *remainders*, respectively, of the input values b_i and the value a the *extractor* of the error-free transformations. Since all contributions q_i are integer multiples of the same power of two, and that their sum is representable with the same format, their floating-point summation is free of rounding error and thus order-independent: $q_1 \oplus q_2 \oplus q_3 = q_1 + q_2 + q_3 = 265$.

While this procedure solves the problem of order-independent bit-reproducibility, it has two problems: First, it only works under the assumption that the absolute value of every intermediate result of this summation, including the final result, is strictly bounded by $2 \cdot \text{ufp}(a)$. An obvious, yet suboptimal solution would be to do two passes over the data, the first one computing the maximum absolute value for which all assumptions are fulfilled, the second one for the actual summation. Second, the sum is inaccurate because relevant parts of the input values, namely the remainders, are discarded. We present an algorithm that solves both problems.

C. Accurate Reproducible Scalar Summation

In HPC, the problem of summing up long vectors of numbers has been studied in detail. In this section, we explain RSUM [14]. In the subsequent section, we discuss why it does not work well with SQL GROUPBY.

In order to address the problem of accuracy, we recall that the error-free transformation produces two outputs: the *contribution* q and the *remainder* r . In the previous section, we made use of the contributions to obtain an order-independent summation. Now we make use of the remainders to improve the accuracy of this summation: we perform an error-free transformation on the remainder, this time using the smaller extractor $a^{(2)} := 2^{-W} \cdot a$ (with $W \in \mathbb{N} \setminus \{0\}$). We thus obtain the second-level contribution and remainder of each input value: $q_i^{(2)} := (a^{(2)} \oplus r_i) \ominus a^{(2)}$, $r_i^{(2)} := r_i \ominus q_i^{(2)}$. These second-level contributions can be summed up to obtain a second-level result: $Q^{(1)} := \sum_{i=1}^n q_i$, $Q^{(2)} := \sum_{i=1}^n q_i^{(2)}$. As

we show in Section VI-B, the final result $Q := Q^{(1)} \oplus Q^{(2)}$ is of comparable accuracy as a standard, non-reproducible floating-point summation. If higher accuracy is needed, an arbitrary number of levels L can be used. The value W expressing the logarithm of the ratio of two consecutive extractors $a^{(l)}, a^{(l+1)}$ is bounded by $m - 2$ and it affects the result (the higher, the more accurate) and the cost (the higher, the slower) of the algorithm. Good choices are 18 and 40 for single and double precision respectively and we use these values in this work.

So far, the extractors a are never assumed to be powers of two. The example in Figure 1 shows a power of two as the extractor, but this does not necessarily have to be the case. The only important factor is that the exponent of the extractor never changes, nor do the intermediate results of the error-free transformation. For this reason, the role of the error-free extractor is taken by the running sums $S^{(l)}$ in the algorithm. The running sum will never change its exponent, as is explained later in this section.

We start with the values $S^{(l)} = 1.5 \cdot 2^{f-(l-1) \cdot W}$. The value f can be chosen arbitrarily, as long as the resulting extractor is large enough for the transformation of the first value b_1 , i.e., $f > \log_2 |b_1| + m - W + 1$. Each input value b_i is transformed using $S^{(1)}, S^{(2)}, \dots, S^{(L)}$ as extractors. The resulting contributions $q_i^{(1)}, q_i^{(2)}, \dots, q_i^{(L)}$ are added to $S^{(1)}, S^{(2)}, \dots, S^{(L)}$ respectively. For $|b_i| \geq 2^{W-1} \cdot \text{ulp}(S^{(1)})$, the first level is not large enough to contain its contribution. In this case, the last level $S^{(L)}$ is discarded, all other levels are demoted (e.g., the first-level sum becomes the second-level sum, $S^{(1)} := S^{(2)}$, and the new first-level sum is initialized to $S^{(1)} := 1.5 \cdot 2^W \cdot \text{ulp}(S^{(2)})$. This ensures that all input values can be included in the summation without breaking the assumptions for reproducible results, also avoiding the need for a first pass to find the maximum absolute value in the input.

In order to avoid that the running sums $S^{(l)}$ change their exponent (which would affect the error-free transformation), a check is performed before its usage. $S^{(l)}$ is *usable*, if it lies in the range $[1.5 \cdot \text{ulp}(S^{(l)}), 1.75 \cdot \text{ulp}(S^{(l)})]$. If it does not, a multiple of $0.25 \cdot \text{ulp}(S^{(l)})$ is added to or removed from it, and the corresponding value is subtracted from or added to the carry-bit counter $C^{(l)}$, which is initialized to 0. For instance, if $S^{(l)} = 1.84 \cdot \text{ulp}(S^{(l)})$, then $1 \cdot (0.25 \cdot \text{ulp}(S^{(l)}))$ is subtracted from it, so that the running sum after this operation is $S^{(l)} = 1.59 \cdot \text{ulp}(S^{(l)})$; the value 1 is added to $C^{(l)}$. The complete state of the summation is given by the running sums $S^{(l)}$ and the corresponding carry-bit counts $C^{(l)}$.

Algorithm 2 summarizes this procedure. In this version of the algorithm we assume that the summation state has been initialized. A number of input values are added to this summation state, and its state is stored to main memory again, so that the summation can be resumed later. In order to finalize the summation, the following sum has to be performed:

$$Q := \sum_{l=1}^L \left((S^{(l)} \ominus 1.5 \cdot \text{ulp}(S^{(l)})) \oplus 0.25 \cdot \text{ulp}(S^{(l)}) \cdot C^{(l)} \right) \quad (1)$$

This sum is not order-independent, thus a predefined order

```

1: Load state  $S^{(l)}, C^{(l)} \forall 1 \leq l \leq L$ 
2: for  $i = 1$  to  $n$  do
3:    $\triangleright$  Check extractor validity, update levels if needed:
       necessary
4:   while  $|b_i| \geq 2^{W-1} \cdot \text{ulp}(S^{(1)})$  do
5:     for  $l = L$  to  $2$  do
6:        $S^{(l)} \leftarrow S^{(l-1)}; C^{(l)} \leftarrow C^{(l-1)}$ 
7:        $S^{(1)} \leftarrow 1.5 \cdot 2^W \cdot \text{ulp}(S^{(2)}); C^{(1)} \leftarrow 0$ 
8:      $\triangleright$  Load and transform value  $b_i$ , update  $S^{(l)}$ :
9:      $r_i^{(0)} \leftarrow b_i$ 
10:    for  $l = 1$  to  $L$  do
11:       $q_i^{(l)} \leftarrow (r_i^{(l-1)} \oplus S^{(l)}) \ominus S^{(l)}$ 
12:       $S^{(l)} \leftarrow S^{(l)} \oplus q_i^{(l)}$ 
13:       $r_i^{(l)} \leftarrow r_i^{(l-1)} \ominus q_i^{(l)}$ 
14:     $\triangleright$  Carry-bit propagation:
15:    for  $l = 1$  to  $L$  do
16:      Find  $d \in \mathbb{Z}$  s.t.  $S^{(l)} \ominus d \cdot 0.25 \cdot \text{ulp}(S^{(l)}) \in$ 
         $[1.5 \cdot \text{ulp}(S^{(l)}), 1.75 \cdot \text{ulp}(S^{(l)})]$ 
17:       $S^{(l)} \leftarrow S^{(l)} \ominus d \cdot 0.25 \cdot \text{ulp}(S^{(l)})$ 
18:       $C^{(l)} \leftarrow C^{(l)} \oplus d$ 
19: Store state  $S^{(l)}, C^{(l)} \forall 1 \leq l \leq L$ 

```

Algorithm 2: RSUM SCALAR.

b_1	1.0101	b_2	1001.0	b_3	100.01		
						$S^{(1)}$	1101100.
						$q_1^{(1)}$	-1100000.
						$r_1^{(1)}$	1100.
						$S^{(2)}$	11010.
						$q_2^{(1)}$	-11000.
						$r_2^{(1)}$	10.
						$S^{(1)} \oplus Q^{(2)}$	1110.0

Figure 2: Application of the RSUM algorithm on three values.

has to be imposed. In order to avoid cancellation, we perform it in reverse order, i.e., we start from the last level.

Figure 2 shows the application of Algorithm 2 on the values $b_1 = 1.3125$, $b_2 = 9$, and $b_3 = 4.25$, with the format defined by $m = 4$, the parameter $W = 2$, the first extractor chosen with $f = 4$, and two extraction levels. The figure uses only binary digits. In the first iteration, the value b_1 is added to the first-level running sum $S^{(1)}$ (incrementing it by the contribution $q_1^{(1)}$). The remainder $r_1^{(1)}$ is added to the second-level running sum $S^{(2)}$. In the second iteration, $|b_2| \geq 2^{W-1} \cdot \text{ulp}(S^{(1)})$, thus triggering an adjustment of the levels, shown in the white box: The second-level sum is discarded, the first-level sum is moved to the second level, and a new extractor is set as first level. Then, the extraction is performed normally. The third value does not trigger the level adjustment. Finally, the sum for each level is computed ($Q^{(1)}$ and $Q^{(2)}$) and these values are summed to give the final result $1110_2 = 14$. $C^{(l)}$ variables are never shown in this example because their value is always zero, as $S^{(l)} \in [1.5 \cdot \text{ulp}(S^{(l)}), 1.75 \cdot \text{ulp}(S^{(l)})]$ at all times. Figure 3 illustrates a carry-bit operation on one level



Figure 3: Example of a carry-bit propagation.

when the running sum $S^{(1)}$ has the value 11011_2 , the carry-bit counter $C^{(1)} = 0$, and the value $b_4 = 3.125$ is processed: after the summation the running sum exceeds $1.75 \cdot \text{ufp}(S^{(1)})$. The adjusting number is found to be $d = 1$, and the sum and the carry-bit counter are modified accordingly.

D. Vectorization of the Summation Algorithm

RSUM [14] was originally introduced in a MIMD context, where each process performs the full summation of the local data and the results are finally summed up globally using MPI_Reduce. As a first step to make it suitable to GROUPBY, we propose a SIMD variant of RSUM. In this variant, the running sum of each level is represented in the registers as a tuple of values $\langle S_1^{(l)}, \dots, S_V^{(l)} \rangle$, where V is the width of the register (e.g., for double-precision values on AVX architectures, $V = 4$). Similarly, the carry-bit counts are represented by a tuple of V elements and V input values are transformed and added to the running sums concurrently. Moreover, a tiling optimization is performed: the extractor validity and the carry-bit propagation are performed just once every NB iterations. This is bounded by $NB \leq 2^{-m-W-1}$ [14].

The summation state does not change format: one running sum and one carry-bit count per level are stored in main memory. When loading a summation state from memory into the registers, the first element of the registers is set to the value read from memory (thus, e.g., $S_1^{(1)} = S^{(1)}$), while the other elements of each register are initialized to $1.5 \cdot \text{ufp}(S^{(l)})$ for running sums and to 0 for carry-bit counts. A horizontal summation has to be performed at the end of the algorithm when storing the state to the main memory: The resulting running sum and carry-bit count of each level are:

$$S^{(l)} := 1.5 \cdot \text{ufp}(S_1^{(l)}) \oplus \sum_{v=1}^V \left(S_v^{(l)} \ominus 1.5 \cdot \text{ufp}(S_v^{(l)}) \right), \quad (2)$$

$$C^{(l)} := \sum_{v=1}^V C_v^{(l)} \quad (3)$$

with order-independent sums. Algorithm 3 summarizes this procedure. For brevity and clarity, parts of the algorithm constitute references to the equivalent lines of Algorithm 2.

IV. A REPRODUCIBLE FLOATING-POINT TYPE

As a first solution for reproducible floating-point aggregation with GROUPBY, we propose a data type that can be used as drop-in replacement for intermediate aggregates of floating-point numbers in any state-of-the-art aggregation algorithm with little to no modification.⁷ We base this type on the

⁷The only arithmetic operation that this type supports is addition, so in a real system it would most likely be an internal type of the execution layer not exposed to the user.

- 1: Load state $S_1^{(l)} \leftarrow S^{(l)}$; $C_1^{(l)} \leftarrow C^{(l)} \quad \forall 1 \leq l \leq L$
- 2: $S_v^{(l)} \leftarrow 1.5 \cdot \text{ufp}(S^{(l)})$; $C_v^{(l)} \leftarrow 0 \quad \forall 2 \leq v \leq V, 1 \leq l \leq L$
- 3: **for** $i = 1$ **to** n , **increment by** $V \cdot NB$ **do**
- 4: \triangleright Check $\max_{i \leq j < i+V \cdot NB} |b_j|$, update levels if necessary. See Algorithm 2, lines 3-7
- 5: **for** $j = i$ **to** $i + V \cdot NB$, **increment by** V **do**
- 6: \triangleright Load and transform values $\langle b_j, \dots, b_{j+V-1} \rangle$, update $\langle S_1^{(l)}, \dots, S_V^{(l)} \rangle$. See Algorithm 2, lines 8-13
- 7: \triangleright Carry-bit propagation. See Algorithm 2, lines 14-18
- 8: \triangleright Horizontal summation
- 9: **for** $l = 1$ **to** L **do**
- 10: $S^{(l)} \leftarrow 1.5 \cdot \text{ufp}(S_1^{(l)}) \oplus \sum_{v=1}^V \left(S_v^{(l)} \ominus 1.5 \cdot \text{ufp}(S_v^{(l)}) \right)$
- 11: $C^{(l)} \leftarrow \sum_{v=1}^V \langle C_1^{(l)}, \dots, C_V^{(l)} \rangle$
- 12: Store state $S^{(l)}, C^{(l)} \quad \forall 1 \leq l \leq L$

Algorithm 3: RSUM SIMD.

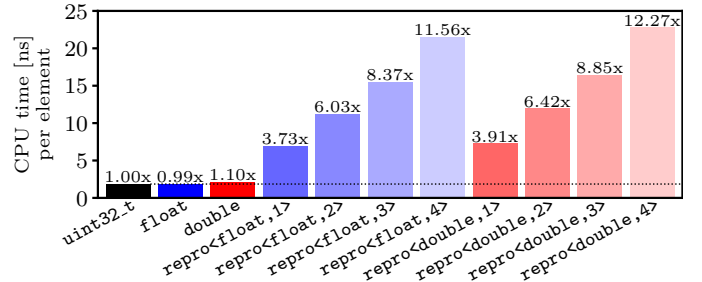


Figure 4: HASHAGGREGATION with different reproducible data types and 16 groups.

reproducible summation algorithm from the previous section: It simply consists of an $\langle \vec{S}, \vec{C} \rangle$ pair, where the symbols $\vec{S} = \langle S^{(1)}, \dots, S^{(L)} \rangle$ and $\vec{C} = \langle C^{(1)}, \dots, C^{(L)} \rangle$ are, respectively, the L levels of the running sum and carry-bit counter as introduced in Section III. In languages such as C++, we can implement this data type as a class with member variables $S[L]$ and $C[L]$ and overload its operator+= for summation with scalars and instances of that type. We refer to this data type as `repro<ScalarT,L>`, where `ScalarT` is either float or double.

Figure 4 shows the performance of a start-of-the-art HASH-AGGREGATION algorithm⁸ instantiated with different variants of the repro data type. This algorithm looks up the aggregate of the corresponding group in a hash table using the `key` field of the input pair and adds the `value` field to that aggregate. We choose the small number of 16 groups to eliminate effects not related to the data types themselves (such as cache effects or pre-processing costs). As the plot shows, the algorithm is between $4 \times$ and $12 \times$ slower with the reproducible data types than with integers or IEEE floats and the more so the more levels of summation we use (i.e., the higher the precision). This

⁸The experimental setup as described in Section VI-A.

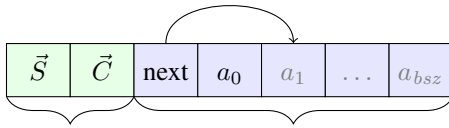


Figure 5: Memory layout of an intermediate aggregate.

is not surprising considering the computational overhead: in the `operator+=` of reproducible types, each level of summation requires about 12 floating-point operations and 4 load and store instructions, while the `operator+=` of standard data types only requires a single one. Finally, there is virtually no difference between single and double precision. This is due to the fact that the algorithm is heavily compute bound and the latency of most instructions does not depend on the operand width.

We have learned two things from this section. First, it requires relatively little development effort to make a large class of algorithms for aggregation with `GROUPBY` bit-reproducible on floating-point numbers. Second, this approach comes at a high price: If we want to match the precision of IEEE floats, we need the types with $L = 2$, which, in the situation shown above, makes the algorithm more than 6 times slower. In the next section, we show a more involved algorithm that improves the slowdown to between $1.9 \times$ and $2.4 \times$ for any `repro<ScalarT,L>`.

V. AGGREGATION WITH SUMMATION BUFFERS

In this section, we present a novel algorithm for aggregation with `GROUPBY` that achieves bit-reproducibility on floating-point numbers efficiently. We first describe the main idea, summation buffers, which allows to use our efficient, vectorized summation routine, and then incorporate that into a state-of-the-art `AGGREGATION` algorithm.

A. Summation Buffers

The main idea for batching the aggregation of input values can be used in any `AGGREGATION` algorithm: we store a reproducible float along with a buffer of input values as intermediate aggregates, which we call *summation buffer*. A summation buffer consists of an array of input values and the offset of the next free slot in the array (“next”). The layout of intermediate aggregates is thus as shown in Figure 5.

For example, the textbook `HASHAGGREGATION` [25] with summation buffers works as follows: Whenever we process a $\langle key, value \rangle$ pair, we first use the key to lookup the entry of the group in the hash table and then use the offset to append the value to the buffer of the group (incrementing the offset accordingly). Only when a buffer is full, we aggregate the buffered values and reset the offset to 0, i.e., to the beginning of the buffer. This allows us to use our vectorized summation algorithm `RSUM SIMD` (Algorithm 3), which is much more efficient than the per-element summation from the previous section. In languages such as C++, we can implement this as new data type again, where the summation operators contain the logic just described, and use this new data type in any existing `AGGREGATION` algorithm transparently.

```

1: partitions  $\leftarrow$  PARALLELPARTITION(input, key,  $F = f^d$ )
2: for each p in partitions with index i parallel do
3:   privateTables[i]  $\leftarrow$  HASHAGGREGATION(p)
4: for each t in privateTables parallel do
5:   for each  $\langle key, value \rangle$  in t do
6:     sharedTable[key] += value

```

Algorithm 4: PARTITIONANDAGGREGATE.

One important tuning parameter is now the buffer size (bsz). On the one hand, the larger the buffer is, the better the constant costs associated with a call to the summation algorithm can be amortized. On the other hand, the larger each buffer is, the larger is also the cache footprint of the algorithm, which may decrease performance significantly. The rest of this section shows how to design an algorithm that makes the best trade-off in all situations.

B. High-Level Algorithm Structure

The overall structure of our `AGGREGATION` algorithm is illustrated in Algorithm 4: We partition the input on the hash value of the keys (Line 1), which can be done very efficiently on modern hardware [9, 26, 33]. Since all input records for a particular group are copied into the same partition, each partition can be processed independently, which we do using `HASHAGGREGATION` (Lines 2 to 3). Finally, we combine the intermediate results into a single hash table shared among all threads (Lines 4 to 6). We call this algorithm `PARTITIONANDAGGREGATE`.

The partitioning effectively divides the number of groups per partition and, hence, the cache footprint of `HASHAGGREGATION` by the partitioning fan-out F , which may outweigh the additional costs for partitioning. Depending on the number of groups in the input, a larger or smaller fan-out is needed in order to fit the working set of `HASHAGGREGATION` into the cache. For a small number of groups, no partitioning may be required. In this case, i.e., if $F = 1$, `PARALLELPARTITION` is a no-op that forwards its input. Since modern hardware can run `PARTITIONING` efficiently only up to a certain fan-out [9, 26, 33], we implement it recursively using zero or more levels of partitioning i.e., we partition with $F = f^d$ for $f = 256$ and $d = 0, 1, \dots$

All phases of the algorithm can be fully parallelized: The partitioning routine called in Line 1 can be parallelized by splitting the input in an arbitrary way (e.g., into equally-sized chunks, using a work queue, or using work stealing) and logically concatenating the corresponding output partitions produced by different threads. After the partitioning, each thread gets a subset of the partitions, which it aggregates into a private hash table independently of the other threads (Lines 2 to 3). With some care, the subsequent transfer to the shared hash table (Lines 4 to 6) can be implemented without synchronization since the threads work on non-overlapping subsets of its content. If no partitioning needs to be done, i.e., if $F = 1$, the shared hash table needs some form of synchronization such as locks. However, since in this case

there are only few groups and each of them only appears once in the hash table of each thread, this last phase takes a negligible amount of time, so the overhead of locking is acceptable.

In order to make this algorithm reproducible, we use summation buffers as the data type for the intermediate aggregates produced by HASHAGGREGATION in Line 3. In the process of aggregating its share of the input or its partitions, each thread calls `operator+=(ScalarT)` on the appropriate intermediate aggregates, which makes it effectively alternate between probing the hash table in order to append input values at the end of their corresponding buffers and summing up the content of buffers as they become full. The shared hash table used in Line 6 has aggregates of type `repro<ScalarT,L>`, i.e., it does not use summation buffers and `operator+=(repro<ScalarT,L>)` is used for merging the intermediate aggregates of the different threads.

The careful reader may wonder why the data of a particular partition is first aggregated into a private hash table and then transferred into a part of the shared hash table that the thread has exclusive access to. It seems possible to save the transfer by aggregating into summation buffers in the shared hash table directly. However, this would have several disadvantages: (1) for all but almost distinct inputs, the transfer is negligible as argued earlier, so there is no need to speed up this phase, (2) the result would consist of summation buffers, which take up more space than needed, and (3) to finalize the computation, we need to iterate over the results anyway in order to flush the buffers.

C. Tuning Buffer Size and Partitioning Depth

PARTITIONANDAGGREGATE with summation buffers has thus two parameters that influence its cache footprint: the size of the summation buffers bsz and the partitioning depth d . We now show how to choose these two parameters.

We first determine the size of the summation buffers given a fixed partitioning depth. Since the access to the summation buffers follows the random pattern given by the hash values of the keys of the input records, the cache footprint of the algorithm consists of the size of the hash table, which we can quantify as $n_{groups} \cdot \text{sizeof}(\text{ScalarT}) \cdot bsz$, where n_{groups} is the number of groups in the input. The buffers should be as large as possible in order to amortize constant costs of calling the summation routine. We thus set the buffer size such that they use the entire cache, which is given by the following equation:

$$bsz = \min \left\{ \begin{array}{l} \lceil |cache| / (n_{groups}/F \cdot \text{sizeof}(\text{ScalarT})) \rceil \\ bsz_{max} \end{array} \right\} \quad (4)$$

where $|cache|$ is the size of the last-level cache corresponding to one thread, bsz_{max} the largest buffer size available in the system, and F the partitioning fan-out.

The optimal number of levels of partitioning d depends on the number of groups: It must be large enough to reduce the number of groups per partition to a point where the subsequent, final level of aggregation can be done in cache. It should not

be larger, otherwise, the partitioning has no benefit and its execution only constitutes overhead. In earlier work [26], we propose to select this depth adaptively: start with a private hash table of fixed size; while the number of groups is lower than the threshold, process all input this way; if and when the threshold is crossed, add a level of partitioning and recurse. This has virtually no overhead, so the resulting runtime essentially corresponds to the optimal partitioning depth for any given input. Since the adaptation mechanism is orthogonal to the topic of this paper and incorporation into our algorithm is only a matter of implementation time, we simply determine the optimal number of levels offline and use that in the remainder of this paper.

D. System Integration

We envision the integration of our algorithm into real systems either as a “fix” for SUM on floating-point numbers or as an alternate aggregate function `RSUM(expression, L)`, which would give the user control on the desired precision.

VI. EXPERIMENTAL EVALUATION

In this section, we show micro-benchmarks that justify design decisions taken in the previous sections, evaluate the performance of our algorithms experimentally, and quantify their impact on end-to-end query performance.

A. Experimental Setup

We run the experiments on a system with 256 GiB RAM and two Intel Xeon E5-2630 v3 CPUs, which belong to the Haswell-EP product line. The CPUs have 8 physical cores each clocked at 2.4 GHz, each with private first and second-level data caches of size 32 KiB and 256 KiB, respectively, as well as a 20 MiB last-level cache shared among all cores. The system runs Debian 8 (jessie) with a Linux kernel v3.18.14. HyperThreading and frequency scaling are switched off.

Unless otherwise mentioned, we use $n = 2^{30}$ $\langle key, value \rangle$ pairs as input, where the key is of type `uint32_t` and the type of the value is as follows: It is of type `ScalarT` if we say that an algorithm runs on `repro<ScalarT,L>` (i.e., it is of type `float` or `double`) and of type `DECIMAL(p)` if we say that the algorithms runs on one of these types. We implement the `DECIMAL` types as built-in integers of size 32, 64, and 128 bit⁹ for $p = 9, 19, 38$, respectively, which is a typical way to implement them. The keys are drawn uniformly at random from the range $[0, n_{groups})$. Due to the nature of random distributions, this means that there are actually less than n_{groups} groups in the input if $n_{groups} \approx n$. We omit experiments on other data distributions as known techniques to handle data skew [11, 26] are orthogonal to the topic of this paper and can be included into our algorithms. All presented numbers are averages of ten identical runs, among which we observed low variance.¹⁰ We express the runtime as “CPU time per element” = $T \cdot P/n$, where T is the total

⁹For 128-bit integers, we use the `__int128` type available in recent versions of GCC on our CPU.

¹⁰The relative standard deviation is mostly below 1% and never above 5%.

running time, $P = 8$ the number of processing elements, and n the number of input elements, which simplifies comparison across different machines.

We implemented our algorithms in C++, which we compile with GCC version 4.9.2. Sporadic tests with newer compilers did not bring significant improvements, probably because we manually force optimizations such as function inlining, loop unrolling, and vectorization wherever beneficial. In order to reduce the impact of the operating system, we pin each thread to a different core of a single socket and allocate and initialize all memory on that socket before the experiment. We refrain from using several sockets in order to avoid NUMA effects, again referring to the fact that NUMA-optimizations are orthogonal to reproducibility.

In experiments not shown in this paper, we compared our baseline implementation to that of Cieslewicz and Ross [11]. Our implementation is somewhat faster, mainly because we use IDENTITYHASHING instead of multiplicative hashing. This is not unrealistic in column stores, where dense ranges are common due to domain encoding. Using a real hash function would make all our algorithms slower by the same constant and thus result in even lower relative overhead of reproducibility. Our implementation of PARTITIONANDAGGREGATE is up to 4 times faster than that of Cieslewicz and Ross [11] because we use the highly-tuned partitioning routine used in other work [9, 31, 33]. Back-of-the-envelope calculations suggest that we achieve the same performance as the implementations used in [26], as well, thereby ensuring our baseline for GROUPBY matches the state of the art.

We also ran experiments with an implementation of SORT-AGGREGATION based on the highly-tuned sorting routines of Balkesen [4], which, as discussed above, can be used to make aggregation reproducible by bringing the input in a deterministic order. Even on integers or built-in floats, this algorithm needs a CPU time per element of over 60 ns, which is $20 \times$ more than our algorithm in the best case and at least $3 \times$ more in any case where $n/n_{groups} < 2^6$, so we did not pursue this approach further.

B. Vectorized Summation Algorithm

We first evaluate accuracy and performance of our summation routine presented in Section III.

1) *Accuracy*: The accuracy of our summation routine RSUM SIMD (Algorithm 3) depends on the number of levels of running sums and carry-bit counters, L . To quantify the accuracy of our routine, we compare its absolute error with that of the non-reproducible summation of conventional floats. According to Demmel and Nguyen [13], the latter error can be bounded by:

$$e_{conv} := (n - 1) \cdot \varepsilon \cdot \sum_{i=1}^n |b_i|, \quad (5)$$

where ε represents a machine constant [21] and b_i , $i = 1..N$, represent the summands. The error of our routine can be

	$n = 10^3$		$n = 10^6$	
	U[1,2)	Exp(1)	U[1,2)	Exp(1)
Conventional	$1.7 \cdot 10^{-10}$	$1.1 \cdot 10^{-10}$	$1.7 \cdot 10^{-4}$	$1.1 \cdot 10^{-4}$
RSUM ($L = 1$)	$1.0 \cdot 10^3$	$1.1 \cdot 10^4$	$1.0 \cdot 10^6$	$1.1 \cdot 10^7$
RSUM ($L = 2$)	$9.1 \cdot 10^{-10}$	$1.0 \cdot 10^{-8}$	$9.1 \cdot 10^{-7}$	$1.0 \cdot 10^{-5}$
RSUM ($L = 3$)	$8.3 \cdot 10^{-22}$	$9.1 \cdot 10^{-21}$	$8.3 \cdot 10^{-19}$	$9.1 \cdot 10^{-18}$

Table II: Maximum absolute error of conventional and reproducible summation algorithms in double precision.

bounded the following expression, which is due to Demmel and Nguyen [14] and the same for their and our algorithm:

$$e_{\text{RSUM SIMD}} := n \cdot 2^{(1-L) \cdot W - 1} \cdot \max_{1 \leq i \leq n} |b_i|, \quad (6)$$

where f and W are the exponent of the first extractor used and the ratio of two consecutive extractors, respectively.

To quantify these expressions in a simple way, we consider the summation of random arrays, with uniformly-distributed values in the range $[1, 2)$ and exponentially-distributed values with $\lambda = 1$, with varying number of values n . With the latter distribution, the probability of an input set with 10^6 values to contain a value larger than 22 is lower than 0.03%, thus we choose 22 as the maximum expected input value and we use it to give a reasonable error bound. Table II shows the expected values of the error bounds for the different algorithms and different distributions in double precision. The error bound for the single-level reproducible summation can be surprisingly large. The reason for such large bounds is the low control on the magnitude of the levels the algorithm has. The largest extractor used for the summation could have a much larger magnitude than the result (up to 2^{W-1} times larger). In this unfortunate case, only one significant bit of the result is kept by the first summation level. If this is the only level used, the uncertainty on the result is as large as the result itself. In many cases, the one-level summation can deliver reasonably accurate results and, for large input sizes, its accuracy can be comparable to the conventional summation. Nevertheless, it gives no guarantee of accurate results beyond the error bounds listed in the table. All error bounds for the reproducible algorithm are up to 2^{W-1} times more pessimistic than the actual error of the summations.

Conclusion: Our summation routine RSUM SIMD with $L = 2$ has comparable accuracy as conventional floating-point summation and achieves much higher accuracy with $L > 2$.

2) *Performance*: We also measure the performance of several variants of the reproducible summation algorithm (RSUM) for summing up a large array of random numbers. Figure 6 shows the result. For RSUM SCALAR and RSUM SIMD (Algorithms 2 and 3, respectively), we sum up the input in chunks of c values for various values of c , i.e., we call the respective algorithm for each chunk of c values. This mimics the pattern of how the summation algorithms are used in the aggregation algorithms of Section V, where they switch between the summation of inputs of different groups. For brevity, the algorithms are simply called SCALAR and SIMD in the figure. We plot the performance in terms of slowdown

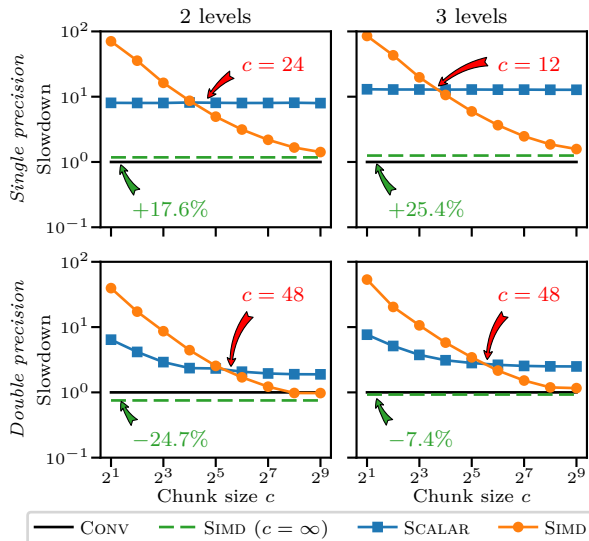


Figure 6: Relative performance of RSUM algorithms compared to a conventional sum using `std::accumulate` (CONV).

compared to a conventional summation algorithm on the same input, which is called CONV in the figure and implemented as a single call to `std::accumulate`. Finally, as “lower bound”, we also plot the performance of a single call to RSUM SIMD, which we call SIMD ($c = \infty$).

As the figure shows, RSUM SIMD is slower than RSUM SCALAR for small chunk sizes, but faster for large chunk sizes. For small chunk sizes, it suffers from a higher start-up overheads because the state it loads and stores from memory into registers and back is a factor V times larger than that of the scalar version. For large chunk sizes, vectorization pays off. The cross-over point (annotated in red) is somewhere between $c = 12$ and $c = 48$ depending on the number of levels L and the precision. As the chunk size reaches $c = 512$, the start-up overhead of the multiple calls to RSUM SIMD is amortized and the performance reaches that of SIMD ($c = \infty$), which consists of a single call. At this point, RSUM SIMD is at most 25% slower than CONV and even somewhat faster in case of double precision. We attribute this to the fact that the compiler is not able to fully vectorize `std::accumulate`, whereas RSUM SIMD ($c = \infty$) is optimized to the point that it is memory-bound.

Conclusion: Our summation routine is faster, the larger the buffer size bsz , as constant start-up costs can be amortized better. For $bsz \geq 2^6$, SIMD is always better than SCALAR and for $bsz \geq 2^9$ or earlier, the difference to the maximum throughput is negligible, where the routine is memory-bound.

C. Aggregation without Summation Buffers

We first present experiments of unmodified state-of-the-art aggregation algorithms, i.e., we do not use aggregation buffers. We compare two categories of data types: (1) The reproducible `repro<ScalarT,L>` types presented in Section IV, which corresponds to making AGGREGATION reproducible with minimal development effort, and (2) DECIMAL(p) with various

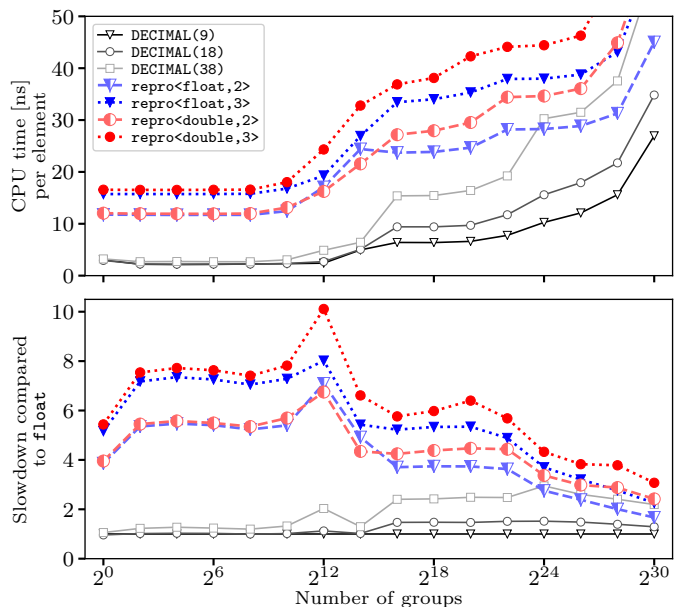


Figure 7: PARTITIONANDAGGREGATE on various `repro<ScalarT,L>` without summation buffers compared to the same algorithm on float/DECIMAL(8).

precisions p (where p is the number of decimal digits), which may be a good enough alternative for some applications. We emphasize that, as discussed in Section II-C, the DECIMAL types are not flexible enough for many modern applications, which cannot determine the scale of the involved numbers statically and thus need a *floating*-point representation. We still include them as a reference point. We also ran the algorithms with built-in floating-point types, but observed exactly the same performance as DECIMAL(9) and DECIMAL(18) for float and double, respectively, so we omit them in the plots shown below.

In order to determine how much partitioning is needed as pre-processing, we use the same procedure as described below in Section VI-D for each data type separately: while for built-in types, using one and two levels pays off starting from 2^{16} and 2^{25} groups, respectively, the thresholds vary somewhat for the reproducible data types. They range from 2^{14} to 2^{16} and 2^{20} to 2^{24} , respectively. This is due to varying amounts of overhead of `operator+=`, which exceeds the costs of PARTITIONING sometimes sooner, sometimes later, but can be easily accommodated for in practice.

Figure 7 summarizes our findings. For better readability, we only show the results for $L = 2$ and $L = 3$, which are the most interesting configurations in practice. As in the experiment shown in Figure 4, the different levels are about equidistant from each other, which gives an idea of the omitted data points.

The upper diagram shows that the runtime for all data types follows the expected pattern: fast, in-cache processing for small numbers of groups and more and more overhead for the partitioning as the number of groups increases, each “step” corresponding to an additional level of partitioning. For DECIMAL(p), the steps are higher the larger p , which is due

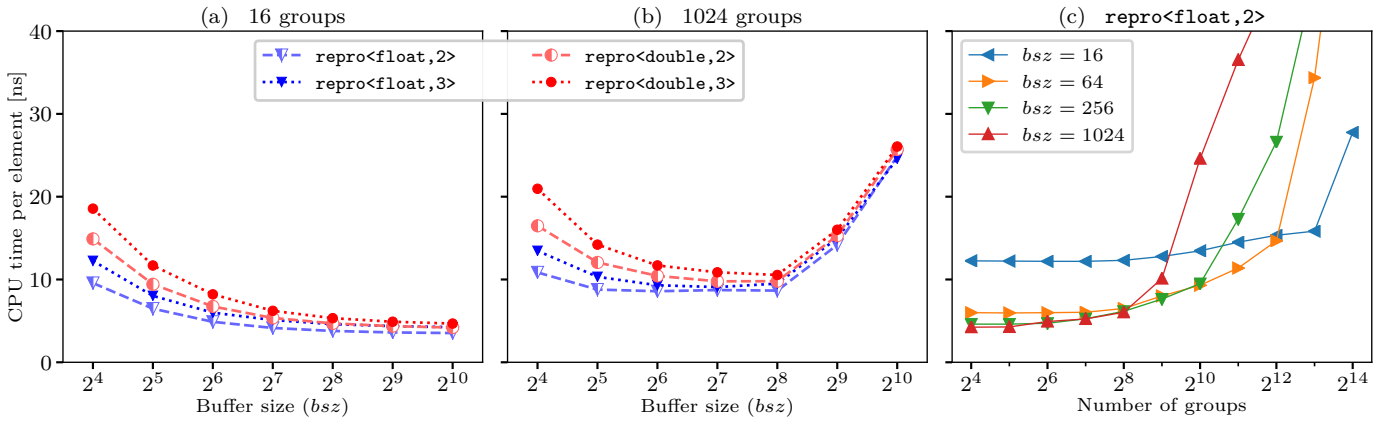


Figure 8: Impact of the buffer size on PARTITIONANDAGGREGATE with $d = 0$.

to the higher memory traffic for wider data types in that phase. As the keys in the input become more and more distinct (for most data types at around 2^{27} groups in the 2^{30} input records), the costs of evicting the final result back to RAM becomes noticeable. Furthermore, the larger L , i.e., the higher the accuracy, the longer the runtime of `repro<ScalarT,L>` by quite a large difference, slightly more so for doubles than for floats. Finally, for `DECIMAL(p)`, the steps due to partitioning are higher the larger p , which is due to the higher memory traffic for wider data types in that phase.

The lower diagram in Figure 7 shows the slowdown of all data types compared to the algorithm on float. As shown earlier, the slowdown of `repro<ScalarT,L>` ranges from a factor¹¹ 4 to 10 for small numbers of groups compared to built-in floats (which has the same performance as `DECIMAL(9)`) and steadily decreases to a factor 1.5 to 3 as the numbers of groups increases. The latter effect is due to the fact that the total runtime increases by the same constant amount for all data types (the partitioning is exactly the same for all of them) with an increasing number of groups, while the overhead of `operator+=` is more or less independent of the number of groups, making the former relatively more dominant than the latter.

Conclusion: Using reproducible floats as drop-in replacement in unmodified state-of-the-art aggregation algorithms has an overhead of up to factor 6 compared to built-in floats with comparable accuracy.

D. Aggregation with Summation Buffers

We now turn our attention to PARTITIONANDAGGREGATE with summation buffers. We first show micro-benchmarks that confirm our choice of the buffer size bsz and then evaluate end-to-end performance.

In order to confirm our model of the cache footprint that leads to Equation 4, we compare the performance of PARTITIONANDAGGREGATE for several values of bsz and $d = 0$, i.e., no partitioning. Figure 8 shows the results. According to Figure 8a and like for the isolated summation routine in

Section VI-B, for a very small number of groups, the larger the buffer is, the better is the performance. However, performance gains of buffers larger than 2^8 elements are only marginal. As Figure 8b shows, the situation is different for slightly larger numbers of groups. Here performance sharply drops for buffers larger than 2^8 and 2^7 elements for single-precision and double-precision data types respectively.

This is in line with our model of the working set introduced in Section V-C: For all configurations shown in Figure 8b, the performance drops when the modeled working set exceeds 1 MiB, which is about half of the last-level cache *per core*. Figure 8c details for which number of groups the caching effect starts to be noticeable for a given buffer size using the `repro<float,2>` data type: For each fixed buffer size, the performance drops sharply once a certain number of groups is reached. Again, this is the point when the working set exceeds 1 MiB according to the equation of Section V-C. The performance of the other data types follows the same pattern.

If we compare the value of bsz predicted by Equation 4 with the buffer size performing best in Figure 8c, we can see that the predicted value is very close to the optimal in all situations, with only a few exceptions (for example, $bsz = 512$ is slightly better than the predicted $bsz = 1024$ for 2^6 groups). However, we found through exhaustive search that 75% of all combinations of number of groups and data types deviate by less than 1% from the optimal performance, 90% of them deviate by less than 5%, and the largest deviation is 20%. Appendix B shows the same procedure for PARTITIONANDAGGREGATE with $d = 1$, where our model predicts the optimal buffer size equally well (the percentiles of the slowdown are even slightly lower).

Conclusion: We can predict a close-to-optimal buffer size using Equation 4, which we use for the remainder of this paper.

In order to determine how much partitioning is needed as pre-processing, we compare PARTITIONANDAGGREGATE for different number of levels of partitioning. Each variant uses the optimal value of bsz as predicted by Equation 4 for the given number of groups. Figure 9 shows the result for `repro<float,2>`: By comparing the performance of the three variants, we can see that each level of partitioning has additional costs,

¹¹For the omitted configuration of $L = 4$, the slowdown is even up to $12 \times$.

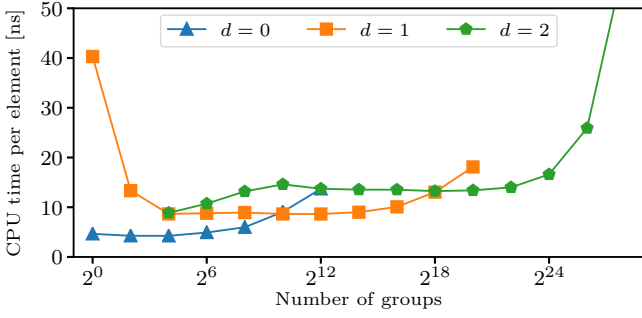


Figure 9: HASHAGGREGATION variants with different amount of partitioning on `repro<float,2>`.

which is only worth it if the number of groups is so large that the inefficiency due to smaller buffers (or out-of-cache processing) is even more expensive. Concretely, no partitioning at all is faster as long as the number of groups is less than 2^{10} . After that point, partitioning once is faster, so the partitioning pays off.¹² Similarly, two levels of partitioning are faster than just one for 2^{18} groups or more. This corresponds to 2^{10} groups per partition—so the two thresholds are effectively the same.

More levels of partitioning are not beneficial for the data sets with 2^{30} records used in this paper, but may be helpful for even larger ones. The performance drop with more than 2^{24} is not due to the number of groups becoming too large, but due to the number of records per group becoming too small. We discuss this effect in more detail in Appendix A.

In experiments not shown in this paper, we determine the thresholds on other data types as well. The results are largely the same, so we omit them here. Furthermore, we only do the tuning with numbers of groups that are powers of two. We could determine the thresholds more precisely, but since small changes in their values do not have a drastic impact on performance, we do not implement that in our prototype.

Conclusion: Partitioning the input helps to reduce the cache footprint of our algorithm, which in turn improves performance. The more groups there are in the input, the more levels of partitioning are needed and our micro-benchmarks show the cross-over points.

Figure 10 shows the performance of PARTITIONANDAGGREGATE using `repro<ScalarT,L>` and summation buffers in comparison with unbuffered DECIMAL types. The upper diagram shows the absolute running time, which exhibits the same pattern of increasing cost due to more levels of partitioning for increasing numbers of groups. Compared to the algorithm without summation buffers, however, the running time is generally lower and, in particular, there is now little difference between different configurations of `repro<ScalarT,L>`. The largest difference is caused by the fact that

¹²If the number of partitions is smaller than the number of threads, some threads are idle while the others aggregate their partitions, which explains the performance drop in these cases. However, the final algorithm uses PARTITIONANDAGGREGATE only for much larger numbers of groups and is, hence, not affected by this phenomenon.

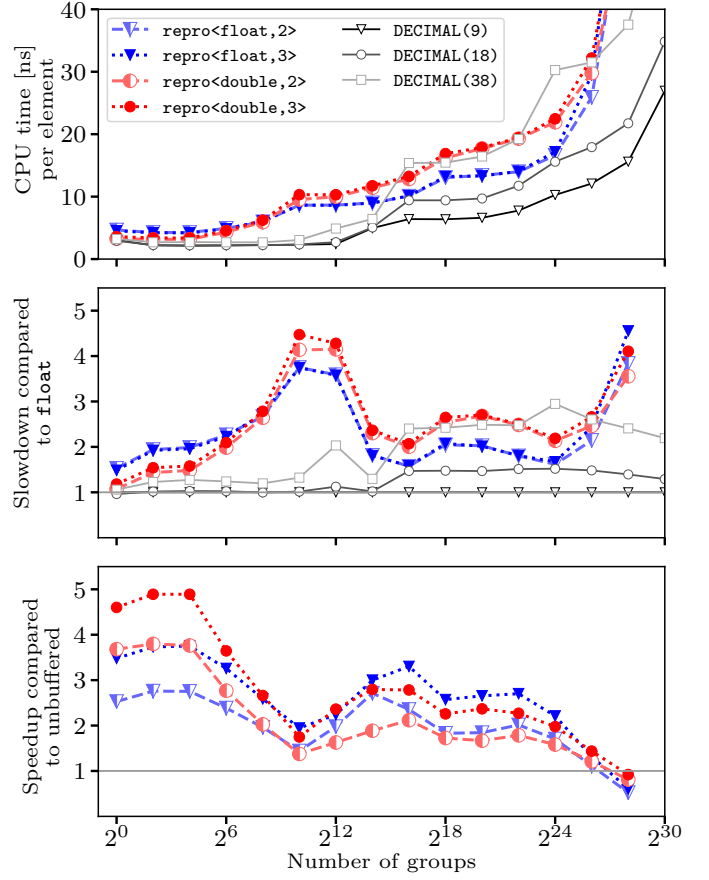


Figure 10: PARTITIONANDAGGREGATE with summation buffers on various `repro<ScalarT,L>` compared to the same algorithm on unbuffered DECIMAL.

the reproducible data types based on double are slower than those based on float. This is mainly due to the fact that PARTITIONING, which is memory bound, needs to move twice as much data in the former case. This is the same effect that slows down the DECIMAL types, which makes them about as slow or slower as our reproducible types for 2^{16} groups and more (in addition to being less flexible). Finally, starting at around 2^{24} groups, where the keys become more and more distinct, performance starts dropping considerably, an effect that we explain in more detail in Appendix A.

If we compare this result with the known results for algorithms on built-in types [11, 26], we see that the algorithms here run out of cache already for much smaller numbers of groups. This is due to the larger cache footprint of the summation buffers as argued earlier. Running out of cache has also a stronger effect caused by the additional indirection (accessing the offset and the end of the buffer may cause two cache misses instead of just one).

The middle diagram shows what the absolute performance means in terms of slowdown compared to built-in floats. We can see that the summation buffers have significantly narrowed the gap between our new reproducible data types and the built-in floating-point types. In many cases, this slowdown is in

data type	slowdown
repro<double,1>	2.12
repro<double,2>	2.18
repro<double,3>	2.29
repro<double,4>	2.41
repro<float,1>	1.88
repro<float,2>	2.11
repro<float,3>	2.16
repro<float,4>	2.35

Table III: Geometric mean of slowdown of summation buffers compared to unbuffered per data type.

the range of 1.3 to 2.5. Only for almost distinct data (more than 2^{26} groups) and for the range of 2^8 to 2^{12} groups, the slowdown may be larger than that. In the latter range, the number of groups are such that the algorithm on built-in floats can still fit its working set into the last-level cache, while the algorithm using summation buffers cannot, and thus needs to pay the additional price of partitioning the input first (on top of the overhead of buffering and more expensive summation).

Table III shows that the slowdown is still reasonable: its geometric mean of all numbers of groups ranges from 1.87 to 2.35 for types based on `float` and from 2.12 to 2.41 for types based on `double`. We believe that this is an affordable price for full reproducibility.

Finally, the lower diagram shows the speedup of our algorithm with summation buffers compared to the naïve approach without them. In particular for small data sets, the speedup is considerable (between factor 2 and more than 5 for the shown configurations and up to factor 6 for the omitted $L = 4$). As expected, it is the higher, the larger L . The speedup drops slightly below 1 for the largest number of groups, i.e., using summation buffers is actually slower than not using them. Since the difference is not large, we leave this as a small open problem.

Conclusion: Thanks to efficient partitioning routines, careful cache-management, and vectorized summation on summation buffers, the overhead of reproducibility on floating-point numbers can be reduced to a slowdown of about a factor of two.

E. End-to-End Query Performance

We integrated our reproducible data types into MonetDB [9] v11.25.23 in order to quantify their impact on end-to-end query performance. To that aim, we modified MonetDB’s aggregation operator for sum on built-in doubles such that it first aggregates its input into a locally allocated array using our reproducible data types (with or without summation buffers) and then copies the result converted to doubles into the result array allocated by the system.¹³ We run a modified TPC-H benchmark as

¹³This does not technically make MonetDB reproducible because it parallelizes query plans as independent subplans on parts of the input whose intermediate results are merged. Our changes make the aggregation operators of each subplan as well as the merging reproducible, but the splitting of the input remains non-deterministic. We argue that this still gives a good approximation of the performance impact. A full integration would require the introduction of a new type, which is a development effort out of the scope of this paper.

	double	repro<d,4> without buffer	repro<d,4> with buffer	double (sorted)
Aggregations	34.2	51.3	38.7	45.1
Other	65.8	63.1	64.0	682.1
Total	100.0	114.4	102.7	727.2

Table IV: CPU time of different approaches for TPC-H Query 1 relative to the total CPU time on built-in doubles in %.

workload where we replaced all `DECIMAL` columns by `DOUBLE`.

Table IV shows the CPU time of different approaches on Query 1 relative to the CPU time of an unmodified MonetDB. As additional baseline, we include the CPU time of modified queries that sort the input to the grouping and aggregation operators, which is the only way to make them reproducible across input permutations without modifying the system.

As the table shows, using `repro<double,4>` without summation buffers takes about 14% longer due to a 50% CPU time increase of the aggregation operators. This increase is lower than the $10\times$ increase observed with our own aggregation operator in Section VI-C due to the slower baseline of MonetDB’s operator, which performs several overflow checks for each input element. With summation buffers, however, the overhead of reproducibility is a negligible 2.7%. Sorting, in contrast, is more than $7\times$ slower, which shows the importance of a numeric solution such as the one we propose.

VII. RELATED WORK

Aggregation with `GROUPBY` on conventional data types is a well understood problem. In recent years, it has been studied extensively for in-memory, multi-core database systems [11, 26, 36, 37]. The focus of that work was contention-free parallelization and cache efficiency. Which strategy is best to achieve the latter goal mainly depends on the size of the result, which directly depends on the number of groups. The consensus [11, 26] is that algorithms similar to `PARTITIONANDAGGREGATE` with various levels of partitioning are best for different numbers of groups, which is why we build on them in this paper. For the case where the result is larger than a private cache, but smaller than the combined shared cache of all threads, Cieslewicz and Ross [11] show that `SHAREDAGGREGATION` may be a better solution than the other two, which uses a shared (lock-free) hash table, at least in the absence of skew. Similar techniques have been proposed for `JOIN` and `SORT` operators [4, 5, 6, 7, 9]. As we show in this paper, these techniques alone are not sufficient for reproducible floating-point numbers.

Since the number of groups is generally not known in advance and hard to estimate, researchers have proposed mechanisms to select the processing strategy adaptively [11, 26], which is possible with minimal overhead. Furthermore, mechanisms for handling data skew in the input or a constrained amount of memory were proposed [11, 25, 26]. These aspects are somewhat independent of reproducibility and the proposed solutions can be applied to our algorithms as well, so we do not go into further detail in this paper.

Variants of SORTAGGREGATION, have not been found competitive by recent studies [4] (except for presorted inputs [11]), mainly due to much higher computational costs and the difficulty to combine early aggregation with vectorization.

There is a line of research in High-Performance Computing that studies the problem of reproducibility of numerical computations using floating-point numbers [3, 13, 14]. However, as argued throughout the paper, the proposed solutions are not applicable to aggregation with GROUPBY. No work on numeric reproducibility in the field of data processing is known to us.

Our work may seem to contradict attempts to speed up query execution either by approximating the computation or reducing the precision. Prominent examples of the first class include DBO [24], BlinkDB [2], and Sample+Seek [18], as well as recent summarization techniques based on the principle of Maximum Entropy [30]. We consider this work somewhat orthogonal to ours since many sampling techniques are based on deterministic pseudo-random number generators and could require a technique similar to ours to be completely reproducible. Maybe more importantly, from a user’s perspective, it is clearly less surprising to get different results from a system that gives approximate answers by design rather than from one that is assumed to be deterministic. Examples of the second class include work on neural networks with 8-bit and even 1-bit precision by Suda et al. [34] and Courbariaux et al. [12], respectively, as well as the low-precision machine learning framework ZipML [38]. However, as discussed in Section II-C, precision and reproducibility are completely orthogonal (our algorithms could be implemented based on lower-precision floating-point types as well).

VIII. SUMMARY AND CONCLUSION

In this paper we have addressed the problem of bit-reproducible aggregation in database systems. The main challenge is that achieving reproducibility is expensive and cannot be efficiently done with existing algorithms for reproducible summation and for GROUPBY. Any naïve combination of existing results in the two areas leads to prohibitive overheads.

The main insights from the work include identifying the bottlenecks that result from the bookkeeping needed to keep track of rounding errors and the effects that it has in cache locality for high cardinality aggregation. Based on these insights, we have proposed ways to extend existing aggregation operators with bit-reproducibility in a way that the resulting overhead is acceptable and comparable to that of conventional aggregation over built-in types.

With these results, we establish the basis for exploring more complex data types and operations inside the database engine providing the same guarantees and meeting the same requirements as those imposed on regular code processing floating-point data. As part of future work we intend to look into operators for machine learning, vector manipulation, and series analysis based on the algorithms presented in this paper.

ACKNOWLEDGMENTS

We thank Eric Sedlar (Oracle Labs) for bringing the problem of reproducibility in the context of database systems to our attention, as well as for valuable feedback on this work. We also thank Lefteris Sidirourgos for his feedback on the integration of our algorithm into MonetDB.

REFERENCES

- [1] ACM US Public Policy Council. *Statement on Algorithmic Transparency and Accountability*. 2017.
- [2] S. Agarwal et al. “BlinkDB: Queries with Bounded Errors and Bounded Response Times on Very Large Data.” In: *EuroSys*. 2013. DOI: [10.1145/2465351.2465355](https://doi.org/10.1145/2465351.2465355).
- [3] A. Arteaga, O. Fuhrer, and T. Hoefler. “Designing Bit-Reproducible Portable High-Performance Applications.” In: *IPDPS* (2014). DOI: [10.1109/IPDPS.2014.127](https://doi.org/10.1109/IPDPS.2014.127).
- [4] Ç. Balkesen. “In-Memory Parallel Join Processing on Multi-Core Processors.” PhD thesis. ETH Zurich, 2014. DOI: [10.3929/ethz-a-010168223](https://doi.org/10.3929/ethz-a-010168223).
- [5] C. Balkesen, J. Teubner, and G. Alonso. “Main-Memory Hash Joins on Multi-Core CPUs : Tuning to the Underlying Hardware.” In: *ICDE*. 2013. DOI: [10.1109/ICDE.2013.6544839](https://doi.org/10.1109/ICDE.2013.6544839).
- [6] C. Balkesen et al. “Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited.” In: *PVLDB*. 2013. DOI: [10.14778/2732219.2732227](https://doi.org/10.14778/2732219.2732227).
- [7] C. Barthels et al. “Distributed Join Algorithms on Thousands of Cores.” In: *PVLDB*. 2017. DOI: [10.14778/3055540.3055545](https://doi.org/10.14778/3055540.3055545).
- [8] G. E. Blelloch et al. “Internally Deterministic Parallel Algorithms Can Be Fast.” In: *PPoPP*. Vol. 47. 8. 2012. DOI: [10.1145/2370036.2145840](https://doi.org/10.1145/2370036.2145840).
- [9] P. A. Boncz, M. L. Kersten, and S. Manegold. “Breaking the Memory Wall in MonetDB.” In: *CACM* 51.12 (2008). DOI: [10.1145/1409360.1409380](https://doi.org/10.1145/1409360.1409380).
- [10] W.-f. Chiang and G. L. Lee. “Determinism and Reproducibility in Large-Scale HPC Systems.” In: *WoDet* (2013).
- [11] J. Cieslewicz and K. Ross. “Adaptive Aggregation on Chip Multiprocessors.” In: *PVLDB*. 2007.
- [12] M. Courbariaux et al. “Binarized Neural Networks: Training Deep Neural Networks with Weights and Activations Constrained to +1 or -1.” In: (Feb. 2016). arXiv: [1602.02830](https://arxiv.org/abs/1602.02830).
- [13] J. Demmel and H. D. Nguyen. “Fast Reproducible Floating-Point Summation.” In: *ARITH*. 2013. DOI: [10.1109/ARITH.2013.9](https://doi.org/10.1109/ARITH.2013.9).
- [14] J. Demmel and H. D. Nguyen. “Parallel Reproducible Summation.” In: *IEEE Trans. Comput.* 64.7 (2015). DOI: [10.1109/TC.2014.2345391](https://doi.org/10.1109/TC.2014.2345391).
- [15] N. Diakopoulos. “Accountability in Algorithmic Decision Making.” In: *CACM* 59.2 (2016). DOI: [10.1145/2844110](https://doi.org/10.1145/2844110).

- [16] N. Diakopoulos and M. Koliska. “Algorithmic Transparency in the News Media.” In: *Digital Journalism* (2016). DOI: [10.1080/21670811.2016.1208053](https://doi.org/10.1080/21670811.2016.1208053).
- [17] W. Dietz et al. “Understanding Integer Overflow in C/C++.” In: *ICSE*. IEEE, 2012. DOI: [10.1109/ICSE.2012.6227142](https://doi.org/10.1109/ICSE.2012.6227142).
- [18] B. Ding et al. “Sample + Seek: Approximating Aggregates with Distribution Precision Guarantee.” In: *SIGMOD*. 2016. DOI: [10.1145/2882903.2915249](https://doi.org/10.1145/2882903.2915249).
- [19] European Parliament and European Council. *General Data Protection Regulation*. 2016. URL: <http://data.europa.eu/eli/reg/2016/679/oj>.
- [20] Federal Trade Commission. *Big Data: A Tool for Inclusion or Exclusion? Understanding the Issues (FTC Report)*. Tech. rep. 2016.
- [21] D. Goldberg. “What Every Computer Scientist Should Know About Floating-Point Arithmetic.” In: *ACM CSUR* 23.1 (1991). DOI: [10.1145/103162.103163](https://doi.org/10.1145/103162.103163).
- [22] J. Gustafson. *The End of Error: Unum Computing*. CRC Press, 2015. ISBN: 9781482239874.
- [23] R. Jacob, T. Lieber, and N. Sitchinava. “On the Complexity of List Ranking in the Parallel External Memory Model.” In: *MFCS* 8635 (2014). DOI: [10.1007/978-3-662-44465-8](https://doi.org/10.1007/978-3-662-44465-8).
- [24] C. Jermaine et al. “Scalable Approximate Query Processing With The DBO Engine.” In: *TODS* 33.4 (2008). DOI: [10.1145/1412331.1412335](https://doi.org/10.1145/1412331.1412335).
- [25] I. Müller. “Engineering Aggregation Operators for Relational In-Memory Database Systems.” PhD thesis. Karlsruhe Institute of Technology, 2016. DOI: [10.5445/IR/1000055675](https://doi.org/10.5445/IR/1000055675).
- [26] I. Müller et al. “Cache-Efficient Aggregation: Hashing Is Sorting.” In: *SIGMOD*. 2015. DOI: [10.1145/2723372.2747644](https://doi.org/10.1145/2723372.2747644).
- [27] NVIDIA Corporation. *CUDA Toolkit v8.0: cuBLAS*. 2016. URL: http://docs.nvidia.com/cuda/cublas/#cublasApi_reproducibility.
- [28] T. Ogita, S. M. Rump, and S. Oishi. “Accurate Sum and Dot Product with Applications.” In: *ICRA* 26.6 (2004). DOI: [10.1109/CACSD.2004.1393867](https://doi.org/10.1109/CACSD.2004.1393867).
- [29] Oracle. *Java Platform, Standard Edition 8: API Specification*. 2016. URL: <https://docs.oracle.com/javase/8/docs/api/java/math/BigDecimal.html>.
- [30] L. Orr, M. Balazinska, and D. Suciu. “Probabilistic Database Summarization for Interactive Data Exploration.” In: *PVLDB*. Vol. 10. 10. 2017. DOI: [10.14778/3115404.3115419](https://doi.org/10.14778/3115404.3115419).
- [31] O. Polychroniou and K. A. Ross. “A Comprehensive Study of Main-Memory Partitioning and its Application to Large-Scale Comparison- and Radix-Sort.” In: *SIGMOD*. 2014. DOI: [10.1145/2588555.2610522](https://doi.org/10.1145/2588555.2610522).
- [32] T. Rosenquist and S. Story. “Using the Intel Math Kernel Library (Intel MKL) and Intel Compilers to Obtain Run-to-Run Numerical Reproducible Results.” In: *The Parallel Universe* 11 (2012).
- [33] F. M. Schuhknecht, P. Khanchandani, and J. Dittrich. “On the Surprising Difficulty of Simple Things: the Case of Radix Partitioning.” In: *PVLDB*. Vol. 8. 9. 2015. DOI: [10.14778/2777598.2777602](https://doi.org/10.14778/2777598.2777602).
- [34] N. Suda et al. “Throughput-Optimized OpenCL-based FPGA Accelerator for Large-Scale Convolutional Neural Networks.” In: *FPGA*. 2016. DOI: [10.1145/2847263.2847276](https://doi.org/10.1145/2847263.2847276).
- [35] *The GNU MPFR Library*. URL: <http://www.mpfr.org/>.
- [36] J. Wen. “Revisiting Aggregation Techniques for Data Intensive Applications.” PhD thesis. University of California, Riverside, 2013. ISBN: 978-1-303-71220-3.
- [37] Y. Ye, K. A. Ross, and N. Vespapunt. “Scalable Aggregation on Multicore Processors.” In: *DaMoN*. 2011. DOI: [10.1145/1995441.1995442](https://doi.org/10.1145/1995441.1995442).
- [38] H. Zhang et al. “The ZipML Framework for Training Models with End-to-End Low Precision: The Cans, the Cannots, and a Little Bit of Deep Learning.” In: *ICML*. 2016.
- [39] D. Zuras et al. “IEEE standard for floating-point arithmetic.” In: *IEEE Std 754-2008* (2008).

APPENDIX

In this appendix, we show two more experiments that allow interested readers to understand our implementation in more depth.

A. Performance of Partition and Aggregate on Almost Distinct Data

Figure 10 in Section VI-D shows a performance drop of PARTITIONANDAGGREGATE for almost distinct data, i.e., when the number of groups approaches the number of records in the input. Here, we explain the reason for this drop and sketch a possible solution. Figure 11 shows the performance of our algorithm for distinct inputs of various sizes. As the plot shows, the performance drops whenever the average number of records per group $n/n_{groups} < 2^6$ independently of the input size. Several effects add up in these situations: First,

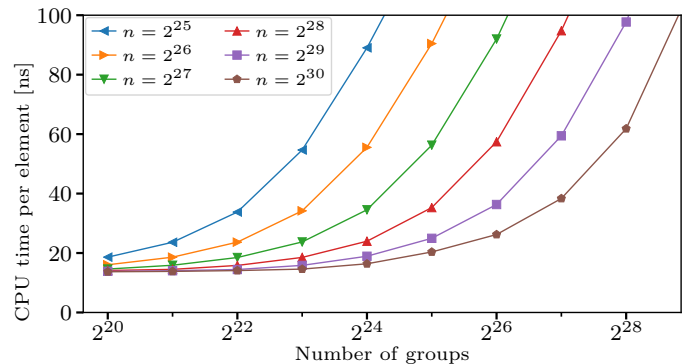


Figure 11: PARTITIONANDAGGREGATE with $bsz = 256$ for various input sizes on `repro<float,2>`.

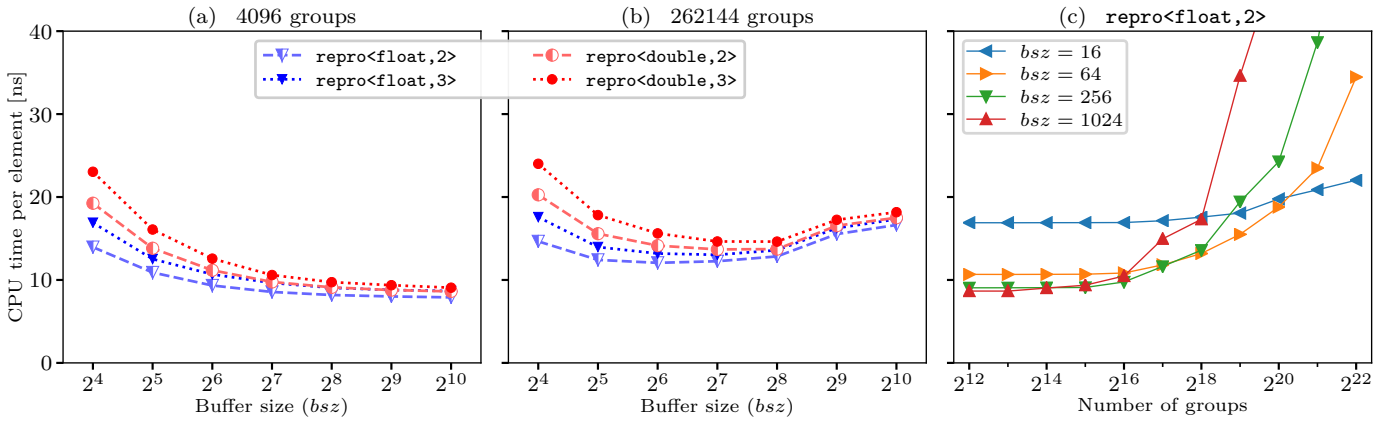


Figure 12: Impact of the buffer size on PARTITIONANDAGGREGATE with a partitioning fanout of 256 and $d = 1$.

the summation routine is less efficient for fewer elements per call as discussed in Section VI-B. Second, as with the algorithm on IEEE floats, the cost of bringing the memory for the final result into cache and writing it back to RAM becomes noticeable as the result size increases. And last, the fact that our reproducible algorithm first produces local aggregates using summation buffers and only then writes them to the result represents additional costs that increase linearly with the number of groups and become dominant after some point. We believe that our algorithm can be improved upon for these cases, but leave such an optimization for future work.

B. Tuning the Buffer Size in Partition and Aggregate

Figure 12 shows the impact of the buffer size (bsz) on the running time of PARTITIONANDAGGREGATE for various numbers of groups in the input. Qualitatively, the impact is exactly the same as without prior partitioning (see Figure 8). However, the partitioning divides the number of groups processed at the same time by the partitioning fan-out (256), i.e., data sets with 256 times more groups can be aggregated before the working set exceeds the cache. At the same time, the running time is increased by the constant costs of the partitioning routine, which is independent of the buffer size or the number of groups.