Reproducible Floating-Point Aggregation in RDBMSs

Ingo Müller
Andrea Arteaga
Torsten Hoefler
Gustavo Alonso

1Systems Group, ETH Zurich

2MeteoSwiss

3Oracle Labs Zurich (past partial affiliation)

ICDE, April 19, 2018
Why reproducibility?

\[ f \to \text{compliance} \]
Why reproducibility?

- Compliance
- Debugging
Why reproducibility?

- compliance
- debugging
- testing
Why reproducibility?

- compliance
- debugging
- checkpointing + fault tolerance
- testing

Reproducible Floating-Point Aggregation in RDBMSs

Alonso, Arteaga, Hoefler, Müller (ETHZ) ICDE'18 2 / 13
Floating-point addition is not associative

CREATE TABLE R (i int, f float);
INSERT INTO R VALUES (1, 2.5e-16);
INSERT INTO R VALUES (2, 0.999...);
INSERT INTO R VALUES (3, 2.5e-16);
SELECT SUM(f) FROM R;

⇝ Returns 0.999...
Floating-point addition is not associative

```sql
CREATE TABLE R (i int, f float);
INSERT INTO R VALUES (1, 2.5e-16);
INSERT INTO R VALUES (2, 0.999...);
INSERT INTO R VALUES (3, 2.5e-16);
SELECT SUM(f) FROM R;
```

\[ \Rightarrow \text{Returns 0.999...} \]

```sql
UPDATE R SET i = i + 1 WHERE i = 2;
SELECT SUM(f) FROM R;
```

\[ \Rightarrow \text{Returns 1.0} \]
Floating-point addition is not associative

```
CREATE TABLE R (i int, f float);
INSERT INTO R VALUES (1, 2.5e-16);
INSERT INTO R VALUES (2, 0.999...);
INSERT INTO R VALUES (3, 2.5e-16);
SELECT SUM(f) FROM R;

⇝ Returns 0.999...
```

```
UPDATE R SET i = i + 1 WHERE i = 2;
SELECT SUM(f) FROM R;

⇝ Returns 1.0
```
Floating-point addition is not associative

CREATE TABLE R (i int, f float);
INSERT INTO R VALUES (1, 2.5e-16);
INSERT INTO R VALUES (2, 0.999...);
INSERT INTO R VALUES (3, 2.5e-16);
SELECT SUM(f) FROM R;

⇝ Returns 0.999...

<table>
<thead>
<tr>
<th>i</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.5e-16</td>
</tr>
</tbody>
</table>

UPDATE R SET i = i + 1 WHERE i = 2;
SELECT SUM(f) FROM R;

⇝ Returns 1.0
Floating-point addition is not associative

CREATE TABLE R (i int, f float);
INSERT INTO R VALUES (1, 2.5e-16);
INSERT INTO R VALUES (2, 0.999...);
INSERT INTO R VALUES (3, 2.5e-16);
SELECT SUM(f) FROM R;

⇝ Returns 0.999...

UPDATE R SET i = i + 1 WHERE i = 2;
SELECT SUM(f) FROM R;

⇝ Returns 1.0
Floating-point addition is not associative

CREATE TABLE R (i int, f float);
INSERT INTO R VALUES (1, 2.5e-16);
INSERT INTO R VALUES (2, 0.999...);
INSERT INTO R VALUES (3, 2.5e-16);
SELECT SUM(f) FROM R;

\[ \Rightarrow \text{Returns 0.999...} \]

UPDATE R SET i = i + 1 WHERE i = 2;
SELECT SUM(f) FROM R;

\[ \Rightarrow \text{Returns 1.0} \]
Floating-point addition is not associative

```
CREATE TABLE R (i int, f float);
INSERT INTO R VALUES (1, 2.5e-16);
INSERT INTO R VALUES (2, 0.999...);
INSERT INTO R VALUES (3, 2.5e-16);
SELECT SUM(f) FROM R;
\[ \Rightarrow \text{Returns } 0.999... \]

UPDATE R SET i = i + 1 WHERE i = 2;
SELECT SUM(f) FROM R;
\[ \Rightarrow \text{Returns } 1.0 \]
```

<table>
<thead>
<tr>
<th>i</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2.5e-16</td>
</tr>
<tr>
<td>2</td>
<td>0.999...</td>
</tr>
<tr>
<td>3</td>
<td>2.5e-16</td>
</tr>
<tr>
<td>3</td>
<td>0.999...</td>
</tr>
</tbody>
</table>
Execution order is affected by many mechanisms

- out-of-place updates
- compression
- indexing
- data aging
- thread scheduling
- degree of parallelism
- ...
Traditional techniques fall short:

- Higher precision  ❌
- Deterministic scheduling  ❌
- Fixed-point arithmetic  ❌
Traditional techniques fall short:

- Higher precision ✗
- Deterministic scheduling ✗
- Fixed-point arithmetic ✗
- Sorting ✗
- Arbitrary precision ✗
Traditional techniques fall short:

- Higher precision ❌
- Deterministic scheduling ❌
- Fixed-point arithmetic ❌
- Sorting ❌
- Arbitrary precision ❌

Numeric methods from HPC:

- Reproducible summation of a vector [DN13; AFH14; DN15]
- Inefficient for grouping ❌
Traditional techniques fall short:

- Higher precision  
- Deterministic scheduling  
- Fixed-point arithmetic  
- Sorting  
- Arbitrary precision

Numeric methods from HPC:

- Reproducible summation of a vector [DN13; AFH14; DN15]  
- Inefficient for grouping

Challenge: integrate a numeric method with low overhead.
Bit-Reproducible Summation

\[ a = \underline{1010} \cdot 1 \]
\[ b = \underline{100} \cdot 1 \]
\[ a + b = \underline{1110} \cdot 1 \]
Bit-Reproducible Summation

\[ a = 1010 . \quad 1010 . \]
\[ b = 100 . 1 \quad 100 . \]

\[ a + b = 1110 . 1 \]
Bit-Reproducible Summation

\[
\begin{align*}
a &= \underline{1010}. \\
b &= \\underline{100}.1 \\
a + b &= \underline{1110}.1
\end{align*}
\]

error-free sum
Bit-Reproducible Summation

\[ \overline{x} := (x + M) - M \]

\[ a = \begin{array}{c} 1010. \\ \end{array} \quad \begin{array}{c} 1010. \\ \end{array} \]

\[ b = \begin{array}{c} 100.1 \\ \end{array} \quad \begin{array}{c} 100. \\ \end{array} \]

\[ a + b = \begin{array}{c} 1110.1 \\ \end{array} \quad \begin{array}{c} 1110. \\ \end{array} \]

error-free sum
Bit-Reproducible Summation

\[ \bar{x} := (x + M) - M \]

\[
\begin{array}{c}
a = 1010. \\
b = 100.1 \\
a + b = 1110.1
\end{array}
\]

error-free sum

- Well-chosen \( M \) makes sum \textbf{associative} (details see paper)
Bit-Reproducible Summation

\[ \bar{x} := (x + M) - M \quad r_x := x - \bar{x} \]

\[
\begin{align*}
a &= \underline{1010.} & = & \underline{1010.} & + & 0.0000 \\
b &= \underline{100.1} & = & \underline{100.} & + & 0.1000 \\
a + b &= \underline{1110.1} & = & \underline{1110.} & + & 0.1000
\end{align*}
\]

**error-free sum**  
**remainders**

- Well-chosen \( M \) makes sum **associative** (details see paper)
Bit-Reproducible Summation

\[
\bar{x} := (x + M) - M \quad \text{and} \quad r_x := x - \bar{x}
\]

- \[ a = \overline{1010}\ . \quad = \overline{1010}\ . \quad + \quad 0\.0000 \]
- \[ b = \overline{100\.1} \quad = \overline{100\.} \quad + \quad 0\.1000 \]
- \[ a + b = \overline{1110\.1} \quad = \overline{1110\.} \quad + \quad 0\.1000 \]

error-free sum \quad remainders

- Well-chosen \( M \) makes sum **associative** (details see paper)
- Sum up **remainders the same way** or drop them \( \rightsquigarrow \) “levels”
Bit-Reproducible Summation

\[
\bar{x} := (x + M) - M \quad \quad r_x := x - \bar{x}
\]

**Example:**

\[
a = \overline{1010.}\quad = \overline{1010.}\quad + \quad 0.0000
\]

\[
b = \overline{100.1}\quad = \overline{100.}\quad + \quad 0.1000
\]

\[
a + b = \overline{1110.1}\quad = \overline{1110.}\quad + \quad 0.1000
\]

**Error-free sum**

**Remainders**

- Well-chosen \( M \) makes sum associative (details see paper)
- Sum up remainders the same way or drop them \( \sim \) “levels”

**Sum can be made associative with \( O(\text{num}_\text{levels}) \) instructions.**
Drop-in Reproducible Floating-Point Numbers

Reproducible Floating-Point Aggregation in RDBMSs
Drop-in Reproducible Floating-Point Numbers

Reproducible Floating-Point Aggregation in RDBMSs

Alonso, Arteaga, Hoefler, Müller (ETHZ) ICDE'18 7 / 13
Drop-in Reproducible Floating-Point Numbers

Drop-in replacement incurs **4-12x slowdown.**
What is the problem?

Input:

| (a, 7) | (b, 3) | (a, 3) | (c, 1) | (b, 1) | (b, 4) | (a, 2) | (b, 5) |

Output (hash table):

(\(a, 7\))
What is the problem?

<table>
<thead>
<tr>
<th>Input:</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(a, 7)</td>
<td>(b, 3)</td>
</tr>
<tr>
<td>(a, 3)</td>
<td>(c, 1)</td>
</tr>
<tr>
<td>(b, 1)</td>
<td>(b, 4)</td>
</tr>
<tr>
<td>(a, 2)</td>
<td>(b, 5)</td>
</tr>
</tbody>
</table>

Output (hash table):

<table>
<thead>
<tr>
<th>(a, 7)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(b, 3)</td>
</tr>
</tbody>
</table>
What is the problem?

Input: 

\[(a, 7) \quad (b, 3) \quad (a, 3) \quad (c, 1) \quad (b, 1) \quad (b, 4) \quad (a, 2) \quad (b, 5)\]

Output (hash table):

\[(a, 7 + 3)\]

\[(b, 3)\]

Startup overhead makes switching between groups costly.
What is the problem?

Input:

(a, 7)  (b, 3)  (a, 3)  (c, 1)  (b, 1)  (b, 4)  (a, 2)  (b, 5)

Output (hash table):

(c, 1)

(a, 7 + 3)

(b, 3)
What is the problem?

Input:

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>(a, 7)</td>
<td>(b, 3)</td>
<td>(a, 3)</td>
<td>(c, 1)</td>
<td>(b, 1)</td>
<td>(b, 4)</td>
</tr>
<tr>
<td>(a, 2)</td>
<td>(b, 5)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Output (hash table):

1. (c, 1)
2. (a, 7 + 3)
3. (b, 3 + 1)

Startup overhead makes switching between groups costly.
What is the problem?

Input: 
(a, 7)  (b, 3)  (a, 3)  (c, 1)  (b, 1)  (b, 4)  (a, 2)  (b, 5)

Output (hash table):

(c, 1)
(a, 7 + 3 + 2)
(b, 3 + 1 + 4)

Startup overhead makes switching between groups costly.
What is the problem?

Input:

| (a, 7) | (b, 3) | (a, 3) | (c, 1) | (b, 1) | (b, 4) | (a, 2) | (b, 5) |

Output (hash table):

| (c, 1) |
| (a, 7 + 3 + 2) |
| (b, 3 + 1 + 4 + 5) |
What is the problem?

Input:

\[(a, 7) \quad (b, 3) \quad (a, 3) \quad (c, 1) \quad (b, 1) \quad (b, 4) \quad (a, 2) \quad (b, 5)\]

Output (hash table):

\[(c, 1)\]

\[(a, 7 + 3 + 2)\]

\[(b, 3 + 1 + 4 + 5)\]
What is the problem?

Input:

| (a, 7) | (b, 3) | (a, 3) | (c, 1) | (b, 1) | (b, 4) | (a, 2) | (b, 5) |

Output (hash table):

| (c, 1) |
| (a, 7 + 3 + 2) |
| (b, 3 + 1 + 4 + 5) |

Startup overhead makes **switching between groups costly**.
Solution: Summation Buffers

Extend hash table entries with buffer:

Hash table entry: key M A c a₀ a₁ a₂ a₃
Solution: Summation Buffers

Extend hash table entries with buffer:

Hash table entry:

| key | M | A | c | a₀ | a₁ | a₂ | a₃ |

Advantages:
- **Amortize** loading and storing state
- **Vectorize** summation
Solution: Summation Buffers

Extend hash table entries with **buffer**:

Hash table entry:  

| key | M | A | c | a₀ | a₁ | a₂ | a₃ |

Advantages:

- **Amortize** loading and storing state
- **Vectorize** summation

Details in the paper:

- How to tune buffer **size**
- How to tune **number** of buffers (through partitioning)
Evaluation: Microbenchmark

- **Number of groups**: 1, 2, 3, 4
- **Slowdown over built-in type**: 5 levels: 1, 2, 3, 4

- **Aggregation buffers**: reduce slowdown to acceptable 2x.
Evaluation: Microbenchmark

The graph shows the slowdown over the built-in type for different numbers of groups and levels. Aggregation buffers reduce the slowdown to acceptable levels.

Number of groups: $2^0, 2^6, 2^{12}, 2^{18}, 2^{24}$

Number of levels: 1, 2, 3, 4

Slowdown over built-in type
Evaluation: Microbenchmark

Number of groups

Slowdown over built-in type

Number of levels

1 2 3 4

Aggregation buffers **reduce slowdown** to acceptable **2x**.
Evaluation: TPC-H Q1 in MonetDB

Our numeric method is significantly faster than sorting.
Evaluation: TPC-H Q1 in MonetDB

Our numeric method is significantly faster than sorting.
Evaluation: TPC-H Q1 in MonetDB

Our numeric method is significantly faster than sorting.
Our numeric method is **significantly faster** than sorting.
Summary

- **Floating-point** numbers are **not reproducible** in current systems.
- **Numeric methods** can help, but have overheads.
- **Summation buffers** amortize overheads and allow vectorization.
