# Cache-Efficient Aggregation: Hashing Is Sorting

Anton Ni

Paper by Müller, I., Sanders, P., Lacurie, A., Lehner, W., and Färber, F.

6.506 Paper Presentation

April 24, 2023

# Motivation

- Grouping with Aggregation is one of the most computationally expensive relational database operators.
- Dominant cost is movement of data.
- We want to reduce accesses to slower main memory.

# Motivation

- Grouping with Aggregation is one of the most computationally expensive relational database operators.
- Dominant cost is movement of data.
- We want to reduce accesses to slower main memory.
- How can an aggregation operator be designed to be cache-efficient?

# Data Aggregation

Input: database with $N$ rows and $C$ columns

General Operation: Group by a subset $S$ of columns and perform some aggregate function on the collection of rows that share the same $S$

# Data Aggregation

General Operation: Group by a subset $S$ of groups and perform some aggregate function on the collection of rows that share the same $S$.

## Example

Input:

| Student | Number of Classes | Hours of Sleep |
|---|---|---|
| A | 4 | 8 |
| B | 5 | 6 |
| C | 4 | 6 |
| D | 5 | 6 |
| E | 3 | 9 |
| F | 3 | 7 |

Group then Average:

| Number of Classes | Avg. Hours of Sleep |
|---|---|
| 4 | 7 |
| 5 | 6 |
| 3 | 8 |

# Two Approaches to Aggregation

Sorting Approach:

1. Sort by grouping attributes.
2. Aggregate consecutive rows of each group.

# Two Approaches to Aggregation

Sorting Approach:

1. Sort by grouping attributes.
2. Aggregate consecutive rows of each group.

Hashing Approach:

1. Using group attributes as the key, place rows into hash table.
2. Aggregate remaining attributes in place.

# Hashing vs. Sorting

## Example

Input:

| Student | Number of Classes | Hours of Sleep |
|---|---|---|
| A | 4 | 8 |
| B | 5 | 6 |
| C | 4 | 6 |
| D | 5 | 6 |
| E | 3 | 9 |
| F | 3 | 7 |

Sort:

| Student | Number of Classes | Hours of Sleep |
|---|---|---|
| E | 3 | 9 |
| F | 3 | 7 |
| A | 4 | 8 |
| C | 4 | 6 |
| B | 5 | 6 |
| D | 5 | 6 |

Hashing:

| Key | Value |
|---|---|
| 4 | 8 |
| 5 | 6 |

# Hashing vs. Sorting

## Example

Input:

| Student | Number of Classes | Hours of Sleep |
|---|---|---|
| A | 4 | 8 |
| B | 5 | 6 |
| C | 4 | 6 |
| D | 5 | 6 |
| E | 3 | 9 |
| F | 3 | 7 |

Sort:

| Student | Number of Classes | Hours of Sleep |
|---|---|---|
| E | 3 | 9 |
| F | 3 | 7 |
| A | 4 | 8 |
| C | 4 | 6 |
| B | 5 | 6 |
| D | 5 | 6 |

Hashing:

| Key | Value |
|---|---|
| 4 | 7 |
| 5 | 6 |

# Hashing vs. Sorting

## Example

Input:

| Student | Number of Classes | Hours of Sleep |
|---|---|---|
| A | 4 | 8 |
| B | 5 | 6 |
| C | 4 | 6 |
| D | 5 | 6 |
| E | 3 | 9 |
| F | 3 | 7 |

Sort:

| Student | Number of Classes | Hours of Sleep |
|---|---|---|
| E | 3 | 9 |
| F | 3 | 7 |
| A | 4 | 8 |
| C | 4 | 6 |
| B | 5 | 6 |
| D | 5 | 6 |

Hashing:

| Key | Value |
|---|---|
| 4 | 7 |
| 5 | 6 |
| 3 | 8 |

# Hashing vs. Sorting

| **Hashing** | **Sorting** |
| --- | --- |
| "Sorts" by hash value | Sorts values |
| Early aggregation | No early aggregation |
| Better if groups are small to fit in cache | Better if number of groups is large |

# Hashing is Sorting

**Claim**

Data aggregation by hashing and sorting are the same in terms of data movement (cache line transfers) following two optimizations.

# Hashing is Sorting

**Claim**

Data aggregation by hashing and sorting are the same in terms of data movement (cache line transfers) following two optimizations.

Authors prove this both by analyzing the cache-line transfers for both paradigms and designing a framework which allows switching between hashing and sorting during execution.

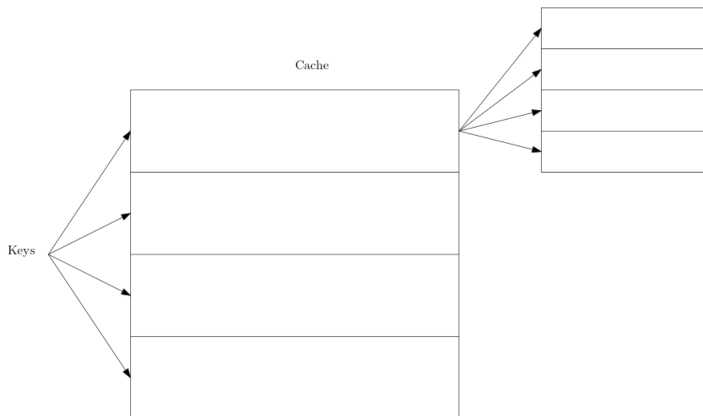
# Analysis of External Aggregation

Assume External-Memory (I/O) Model and the following variables:

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

Costs of an algorithm is just the number of cache line transfers in the worst case.
Output has size $K$.

# Sort-Based Aggregation



# Sort-Based Aggregation

Idea: Use bucket sort to recursively partition input into buckets until data is sorted.

# Sort-Based Aggregation

Idea: Use bucket sort to recursively partition input into buckets until data is sorted.

Recursion stops when buckets are size $B$ since we can sort "for free" within a cache line.

$$\text{Number of Leaves in Recursion Tree} = \frac{N}{B}$$

# Sort-Based Aggregation

Idea: Use bucket sort to recursively partition input into buckets until data is sorted.

Recursion stops when buckets are size $B$ since we can sort "for free" within a cache line.

$$\text{Number of Leaves in Recursion Tree} = \frac{N}{B}$$

Tree has degree $\frac{M}{B}$ since number of partitions is limited by number of cache lines.

# Sort-Based Aggregation

Assuming roughly balanced sorting tree, height is $\left\lceil \log_{M/B} \frac{N}{B} \right\rceil$.

# Sort-Based Aggregation

Assuming roughly balanced sorting tree, height is $\left\lceil \log_{M/B} \frac{N}{B} \right\rceil$. Input is read and written one time each per level of the tree. Then there is one aggregation pass and one write to output.

Approximation for cost:

$$2 \cdot \frac{N}{B} \cdot \left\lceil \log_{M/B} \frac{N}{B} \right\rceil + \frac{N}{B} + \frac{K}{B}$$

# Sort-Based Aggregation

Assuming roughly balanced sorting tree, height is $\left\lceil \log_{M/B} \frac{N}{B} \right\rceil$.
Input is read and written one time each per level of the tree. Then there is one aggregation pass and one write to output.

Approximation for cost:

$$2 \cdot \frac{N}{B} \cdot \left\lceil \log_{M/B} \frac{N}{B} \right\rceil + \frac{N}{B} + \frac{K}{B}$$

assuming static depth of call tree.
Now we relax this assumption by using the fact that recursion stops earlier when $K < N$.

# Sort-Based Aggregation Analysis Optimization

When $K < N$, recursion stops earlier, so cost is:

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

Well known lower bound for multiset sorting.

# Sort-Based Aggregation Analysis Optimization

When $K < N$, recursion stops earlier, so cost is:

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

Well known lower bound for multiset sorting.

What if we merge the last sort pass with final aggregation pass? Instead of writing to memory when the buffer of a partition runs full, aggregate to make space. Results in only $K/B$ leaves. Optimized cost:

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

Note that intermediate results must be $O(1)$ which is usually true.

# Hash-Based Aggregation

Hash-Based Aggregation:
We need $K/B$ cache lines to write, $N/B$ cache lines to read. If $K > N$, only $M/K$ proportion of rows can be in the cache at any time, so 2 cache line transfers for each other row. We get that the cost is:

$$\frac{N}{B} + \begin{cases} K/B & \text{if } K < M \\ 2N(1 - M/K) & \text{otherwise} \end{cases}$$

Very good efficiency when $K$ fits into cache, very poor otherwise.

# Hash-Based Aggregation Optimization

Optimization: recursively partition input by value and apply hash aggregation on each group separately. This reduces the effective $K$, algorithm works in cache. We now have additional costs from partitioning analogous to sort-based aggregation. With $\left(\left\lceil \log_{M/B} \frac{K}{B} \right\rceil - 1\right)$ partitioning passes, cost becomes

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

which is the same as optimized sorting aggregation.

# Hash-Based Aggregation Optimization

Optimization: recursively partition input by value and apply hash aggregation on each group separately. This reduces the effective $K$, algorithm works in cache. We now have additional costs from partitioning analogous to sort-based aggregation. With $\left(\left\lceil \log_{M/B} \frac{K}{B} \right\rceil - 1\right)$ partitioning passes, cost becomes

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

which is the same as optimized sorting aggregation.

*No such duality between hashing and sorting!*

Question: How to engineer a single aggregation algorithm similar to optimized versions of both aggregation algorithms?

# Mixing Hashing and Sorting

Partitioning Routines:

**Algorithm 1** Algorithmic Building Blocks

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

Both functions partition by hash value.
Partition: simple partition by hash value
Hashing: starts with hash table size of cache and replaces the current hash table with a new one every time it is filled

# Aggregation Framework

**Algorithm 2** Aggregation Framework

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

1. Split input into runs.
2. Process input either by hashing or partitioning.
3. Each step of recursive partitioning has more and more hash digits in common within a bucket.

# Aggregation Framework

Some Key Features:

- Framework supports hashing and partitioning interchangeably.
- Hashing is used when we can exploit locality of groups.
- Aggregation can be performed at all levels of recursion.

Aggregation is in a way similar to semi-sorting.

# Parallelization Details

All phases of algorithm can be fully parallelized.

**Algorithm 2** Aggregation Framework

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

Line 5 can be performed in parallel.
Minimial synchronization is needed for unions on line 8.

# Minimizing Computations

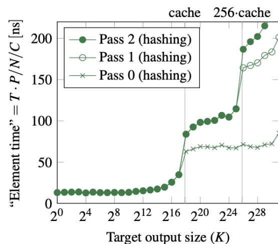
Hash table optimization:

- single hash table with linear probing
- size equal to L3 cache
- collisions rare enough to not affect runtime

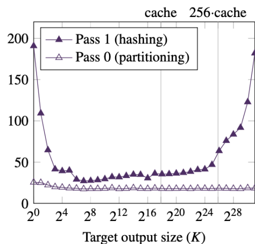
Partitioning optimization:

- “software write-combining” to reduce read-before-write overhead and TLB misses
- data structure which eliminates a counting pass to determine output positions and offsets

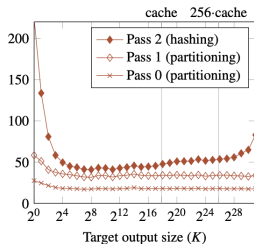
# Hashing or Sorting?



(a) HashingOnly



(b) PartitionAlways (2 passes)



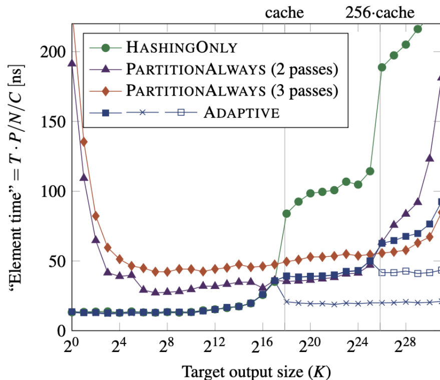
(c) PartitionAlways (3 passes)

General strategy: If $K > N$ and data is uniform (cannot exploit data locality), partition first. Otherwise, use hashing.

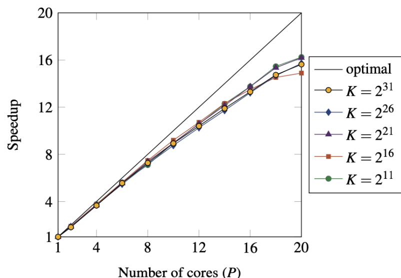
# Adaptive Strategy

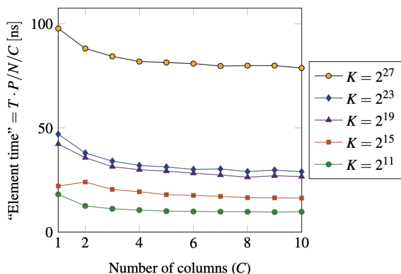
1. Start with hashing.
2. When a hash table gets full, compute $\alpha := \frac{n_{in}}{n_{out}}$.
3. If $\alpha$ is above threshold, continue with hashing.
4. Switch to partitioning once $\alpha$ is below threshold.
5. When enough data has been processed ($n_{in} = c \cdot (\text{cache size})$), try hashing again.

# Hashing or Sorting?



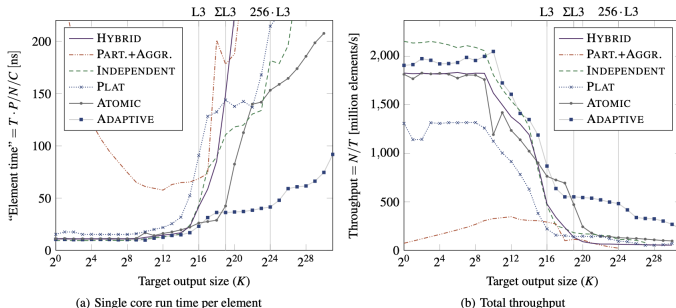
# Scalability



# Comparison with Other Frameworks

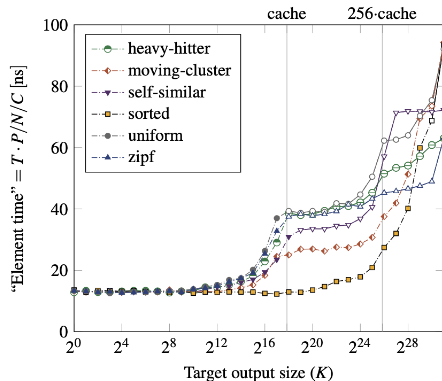
A DISTINCT query with no aggregate columns was used for comparison to abstract from architectural differences.




(a) Single core run time per element

(b) Total throughput

# Skew Resistance



# Conclusion

Summary and Strengths

- Cache line transfers are the main cost in database aggregation algorithms.
- Hashing and Sorting algorithms are equivalent in the external memory model.
- Being able to switch freely between the two protocols is an inherent advantage.
- System reliably outperforms competitors.

Potential Weaknesses:

- A lot of the intermediate aggregation requires $O(1)$ additional space. Unlikely same framework is possible for aggregation computations which require more than $O(1)$ additional space.
- No comparison on groups with more than 1 column were compared with competitors.

# Discussion Questions and Future Research Directions

1. Prove the lower bound on cache line transfers for an aggregation query.
2. Demonstrate runtime advantages with other aggregation functions.
3. How do we improve skew resistance?

# References

I. Müller, P. Sanders, A. Lacurie, W. Lehner, and F. Färber. Cache-Efficient Aggregation: Hashing Is Sorting. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1123–1136, 2015.