CACHE-EFFICIENT AGGREGATION: HASHING *IS* SORTING

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

Presented by Divya Gopinath

6.886 Spring 2019

AGENDA

- Motivation
- Duality of hashing and sorting
- Efficient hashing and sorting primitives
- Design and analysis of aggregation algorithm
- Implementation and optimization of algorithm
- Evaluation and benchmarking
- Discussion

AGENDA

- **Motivation**
- **Duality of hashing and sorting**
- Efficient hashing and sorting primitives
- Design and analysis of aggregation algorithm
- Implementation and optimization of algorithm
- Evaluation and benchmarking
- Discussion

MOTIVATION

- Processing power of multi-core CPUs increasing at a faster rate than memory bandwidth → **I/O complexity is key**
- Relational database operators are expensive, and **aggregation** in particular

```
SELECT column_name(s)
FROM table_name
WHERE condition
GROUP BY column_name(s)
```

  - Limiting factor is movement of data
  - Originally designed to reduce number of disk accesses, main memory accesses considered free
  - Now, move one level up hierarchy to the **cache**

HASHING / SORTING: UNOPTIMIZED

HASHAGGREGATION

- Insert input rows into hash table with grouping attributes as keys
- Aggregate remaining attributes in-place

Use when number of groups is small because output will fit in the cache, and provides early aggregation.

col1	col2
1	3
2	4
1	2
2	5

group_key = h(col1)	sum(col2)
h(1)	3+2
h(2)	4+5

HASHING / SORTING: UNOPTIMIZED

col1	col2
1	3
2	4
1	2
2	5

sort(col1)	col2
1	3
1	2
2	4
2	5

SORTAGGREGATION

- Sorts the rows by grouping attributes
- Aggregates consecutive rows of each group

Use when number of groups is large as hashing isn't as efficient then, but aggregation at later stage.

CLAIM: HASHING *IS* SORTING

Can we optimize hashing/sorting such that cache line transfers are comparable between the two?

Observation 1 : hashing is equivalent to *sorting by hash value*
Intermediate results from hashing can be processed by sorting routine
Hashing makes key domain more dense– an easier sorting problem!

CLAIM: HASHING *IS* SORTING

Can we optimize hashing/sorting such that cache line transfers are comparable between the two?

Observation 1 : hashing is equivalent to *sorting by hash value*
Intermediate results from hashing can be processed by sorting routine
Hashing makes key domain more dense– an easier sorting problem!

Observation 2 : hashing allows us to perform early aggregation
Many repeated keys in distribution? Hash.
Few repeated keys? Sort.

AGENDA

- Motivation
- Duality of hashing and sorting
- **Efficient hashing and sorting primitives**
- Design and analysis of aggregation algorithm
- Implementation and optimization of algorithm
- Evaluation and benchmarking
- Discussion

CLAIM: HASHING *IS* SORTING
IN TERMS OF CACHE COMPLEXITY

External memory model:

- `N = number of input rows`
- `K = number of groups in the input`
- `M = number of rows fitting into cache`
- `B = number of rows per single cache line`

SORT-BASED AGGREGATION

Use bucket sorting to recursively partition and sort input.

We sort each cache line for free before writing, so number of leaves:

$$l = \frac{N}{B}$$

SORT-BASED AGGREGATION

Use bucket sorting to recursively partition and sort input.

We sort each cache line for free before writing, so number of leaves:

$$l = \frac{N}{B}$$

Number of partitions limited by number of buffers that fit into cache, so degree of tree is $\frac{M}{B}$

$$h = \left\lceil \log_{\frac{M}{B}} \frac{N}{B} \right\rceil$$

SORT-BASED AGGREGATION

Use bucket sorting to recursively partition and sort input.

We sort each cache line for free before writing, so number of leaves:

$$l = \frac{N}{B}$$

Number of partitions limited by number of buffers that fit into cache, so degree of tree is $\frac{M}{B}$

$$h = \left\lceil \log_{\frac{M}{B}} \frac{N}{B} \right\rceil$$

Input read and written once per level, and one additional pass to read the input and write the output once, so overall cost:

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

SORT-BASED AGGREGATION

Use bucket sorting to recursively partition and sort input.

We sort each cache line for free before writing, so number of leaves:

$$l = \frac{N}{B}$$

Number of partitions limited by number of buffers that fit into cache, so degree of tree is $\frac{M}{B}$

$$h = \left\lceil \log_{\frac{M}{B}} \frac{N}{B} \right\rceil$$

Input read and written once per level, and one additional pass to read the input and write the output once, so overall cost:

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

In cases where $K < N$, keys form a multiset and recursion stops early, cost slightly lower:

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

This is a lowerbound for multiset sorting.

SORT-BASED AGGREGATION

Use bucket sorting to recursively partition and sort input.

Our current cost:

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

Lastly, merge last bucket sort pass with final aggregation pass…

- Eliminates one pass over entire data
- Hold M partitions instead of M/B
- Only K/B leaves in the call tree
- Intermediate results must be O(1), which is true for `SUM, COUNT, MIN, MAX, AVG`

Total cost:

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

If $K < M$, algorithm reads data once and calculates result in cache.

HASH-BASED AGGREGATION

Hash rows based on attribute we are trying to group by.

We need to write K/B cache lines for result, and read N/B to read input, so long as $K < M$. Cost:

$$\frac{N}{B} + \begin{cases} \frac{K}{B}, & K < M \\ 2\left(1 - \frac{M}{K}\right)N, & o.w. \end{cases}$$

HASH-BASED AGGREGATION

Hash rows based on attribute we are trying to group by.

We need to write K/B cache lines for result, and read N/B to read input, so long as $K < M$. Cost:

$$\frac{N}{B} + \begin{cases} \frac{K}{B}, & K < M \\ 2\left(1 - \frac{M}{K}\right)N, & o.w. \end{cases}$$

Problem: when cache is full, there's a cache miss for almost every input row!

Optimization: partition input and recursively call procedure (each partition reduces K). Now, same number of cache-line transfers as sorting:

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

AGENDA

- Motivation
- Duality of hashing and sorting
- Efficient hashing and sorting primitives
- **Design and analysis of aggregation algorithm**
- Implementation and optimization of algorithm
- Evaluation and benchmarking
- Discussion

COMBINING THE TWO…

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 sort-based and hash-based aggregations use **partition** as a subroutine:

- Partition based on keys of groups
- Partition based on hash value
- Process of building up a hash table also partitions input by hash value!

COMBINING THE TWO…

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 sort-based and hash-based aggregations use **partition** as a subroutine:

- Partition based on keys of groups
- Partition based on hash value
- Process of building up a hash table also partitions input by hash value!

PARTITIONING: one run per partition produced
HASHING: every full hash table split into one run per partition

COMBINING THE TWO…

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)
```

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}, ..., R_{j,F} ← PRODUCERUNS(run, level)
8:      return ⋃_{i=1}^{F} AGGREGATE(⋃_j R_{j,i}, level + 1)
```

Both sort-based and hash-based aggregations use **partition** as a subroutine:

- Partition based on keys of groups
- Partition based on hash value
- Process of building up a hash table also partitions input by hash value!

PARTITIONING: one run per partition produced
HASHING: every full hash table split into one run per partition

- Hash values are partition criterion
- Hashing enables early aggregation (helps in case of locality of groups)
- In absence of locality, use general partitioning
- Similar to a radix sort, as bucket of element determined by bits of a hash function
- Some meta-data to store "super-aggregate" functions, e.g. COUNT vs. SUM

COMBINING THE TWO…

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)
```

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}, ..., R_{j,F} ← PRODUCERUNS(run, level)
8:      return ⋃_{i=1}^{F} AGGREGATE(⋃_j R_{j,i}, level + 1)
```

Both sort-based and hash-based aggregations use **partition** as a subroutine:

- Partition based on keys of groups
- Partition based on hash value
- Process of building up a hash table also partitions input by hash value!

PARTITIONING: one run per partition produced
HASHING: every full hash table split into one run per partition

- Hash values are partition criterion
- Hashing enables early aggregation (helps in case of locality of groups)
- In absence of locality, use general partitioning
- Similar to a radix sort, as bucket of element determined by bits of a hash function
- Some meta-data to store "super-aggregate" functions, e.g. COUNT vs. SUM

Is aggregation just integer sorting? Connection to semisort paper in terms of how we view aggregation as a procedure.

AGENDA

- Motivation
- Duality of hashing and sorting
- Efficient hashing and sorting primitives
- Design and analysis of aggregation algorithm
- **Implementation and optimization of algorithm**
- Evaluation and benchmarking
- Discussion

PARALLELIZATION

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}, ..., R_{j,F} ← PRODUCERUNS(run, level)
8:     return ⋃_{i=1}^{F} AGGREGATE(⋃_j R_{j,i}, level + 1)
```

no I/O shared, so fully parallelizable

negligible synchronization with unions

- Always create parallel tasks for recursive calls
- Work-stealing to parallelize loop over input (robust against heavy skew)
- Some additional work (not discussed in this presentation) to adapt model for two types of storage schemes: column-wise processing vs JiT compilation

MINIMIZING PRIMITIVE COMPUTATIONS

CPU COSTS OF HASHING

- Single-level hash table with linear probing
- Hash table set to size of L3 cache and considered full at 25% fill rate
- Collisions are rare if number of groups much smaller than cache, so no CPU cycles to collision-resolve

MINIMIZING PRIMITIVE COMPUTATIONS

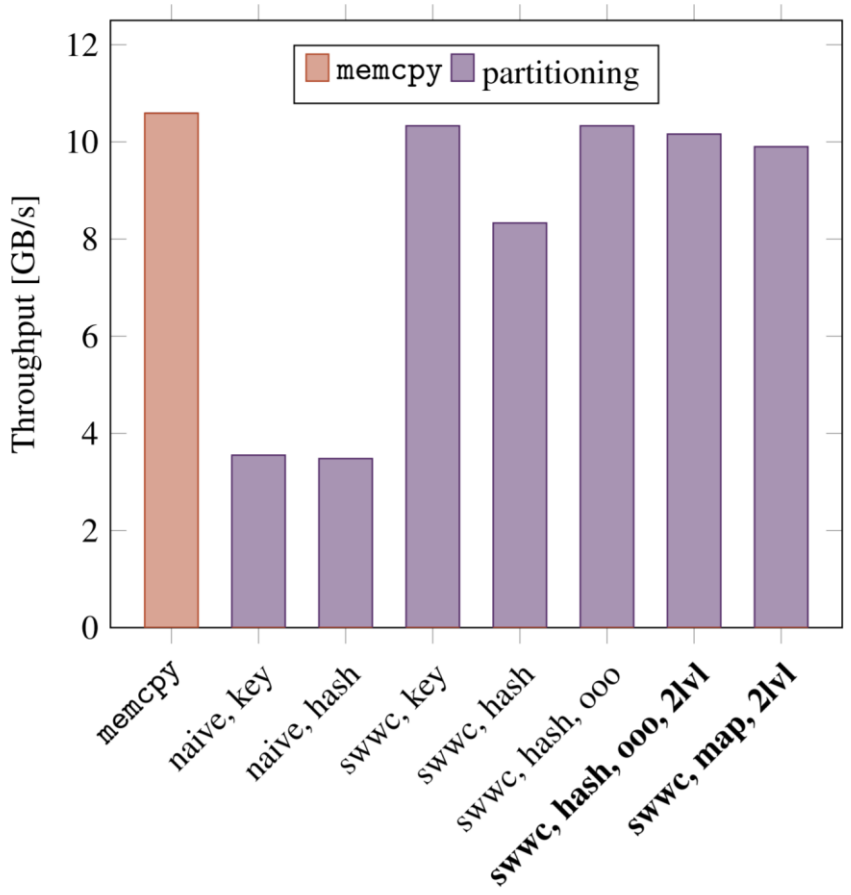
CPU COSTS OF HASHING

- Single-level hash table with linear probing
- Hash table set to size of L3 cache and considered full at 25% fill rate
- Collisions are rare if number of groups much smaller than cache, so no CPU cycles to collision-resolve

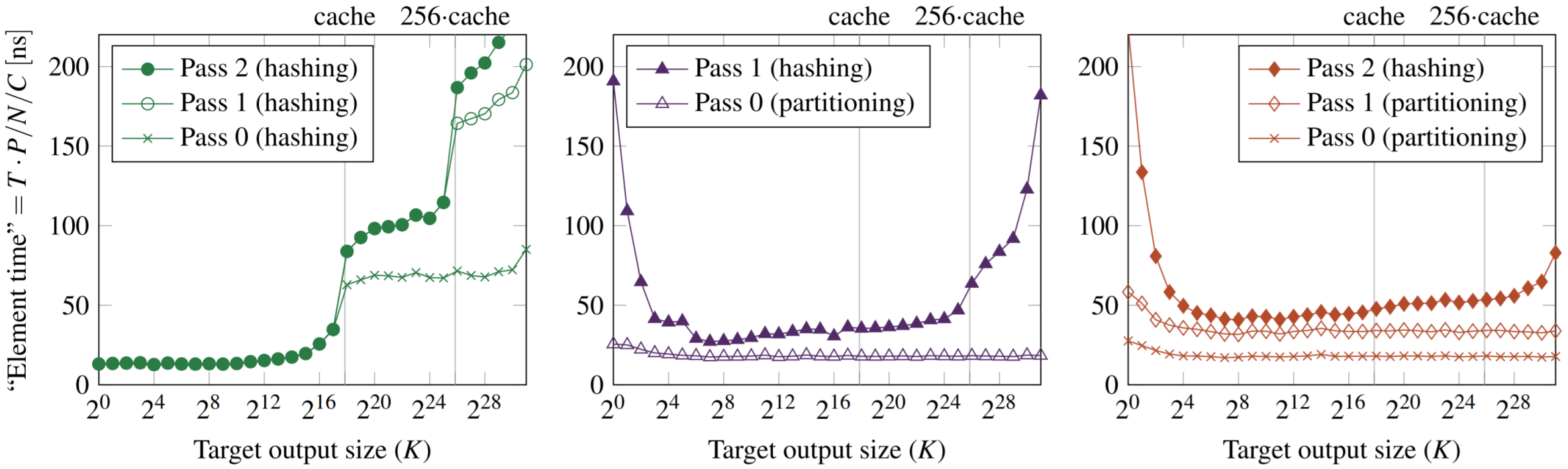
CPU COSTS OF PARTITIONING

- Software-write combining: avoid read-before-write overhead and reduce TLB misses
- Use list of arrays to eliminate counting pass determining output partitions

MINIMIZING PRIMITIVE COMPUTATION



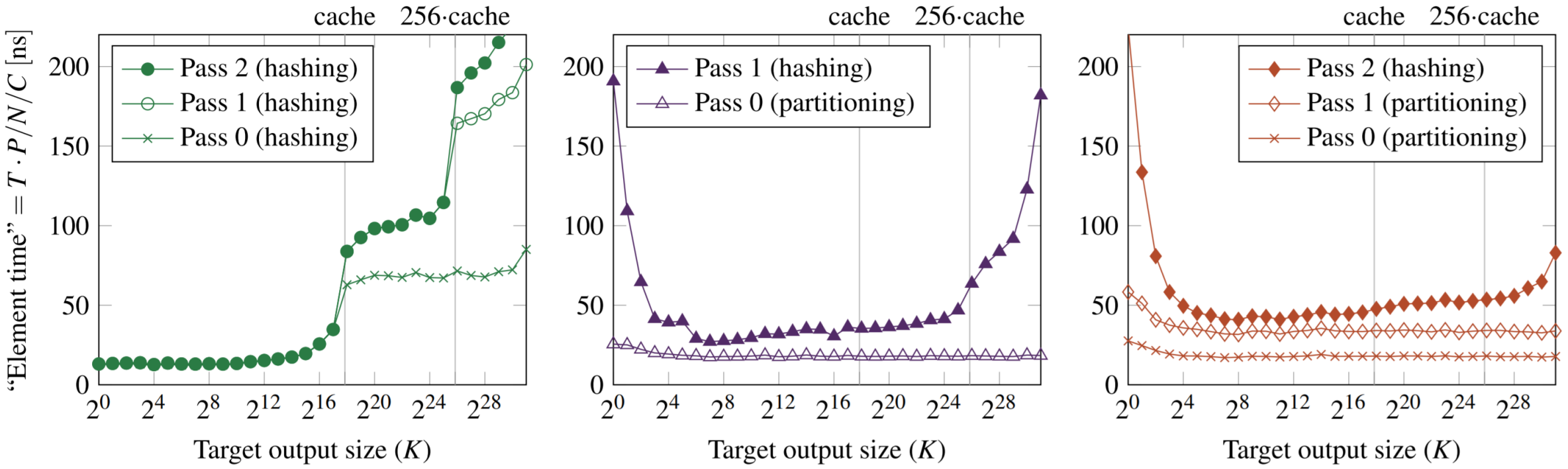
HASHING *IS* SORTING…
BUT WHEN TO PICK BETWEEN THEM?



(a) HASHINGONLY (b) PARTITIONALWAYS (2 passes) (c) PARTITIONALWAYS (3 passes)

1

If the number of groups is smaller than the cache, HASHINGONLY computes the entire result in cache. Otherwise, it recurses until it can. PARTITIONALWAYS does not– doesn't know the right depth to recurse to before hashing pass.



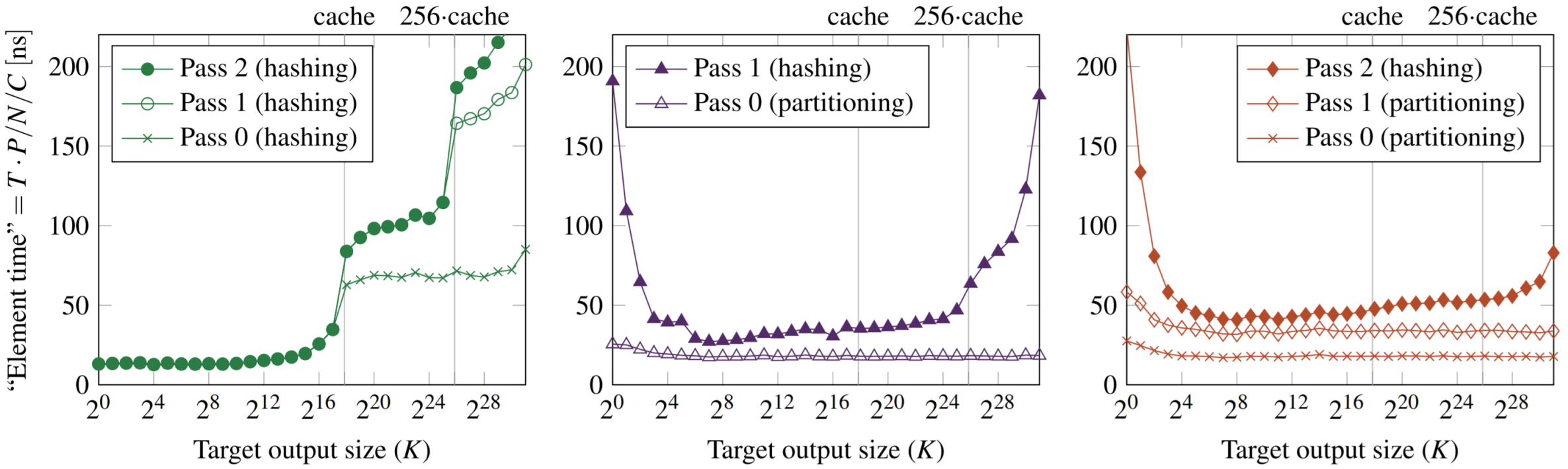
(a) HASHINGONLY (b) PARTITIONALWAYS (2 passes) (c) PARTITIONALWAYS (3 passes)

1

If the number of groups is smaller than the cache, HASHINGONLY computes the entire result in cache. Otherwise, it recurses until it can. PARTITIONALWAYS does not– doesn't know the right depth to recurse to before hashing pass.

2

If K is much bigger than cache, partitioning is much faster. Hashing suffers from non-sequential memory accesses and wasted space. Tuning from before helps partition achieve high throughput.



(a) HASHINGONLY (b) PARTITIONALWAYS (2 passes) (c) PARTITIONALWAYS (3 passes)

1

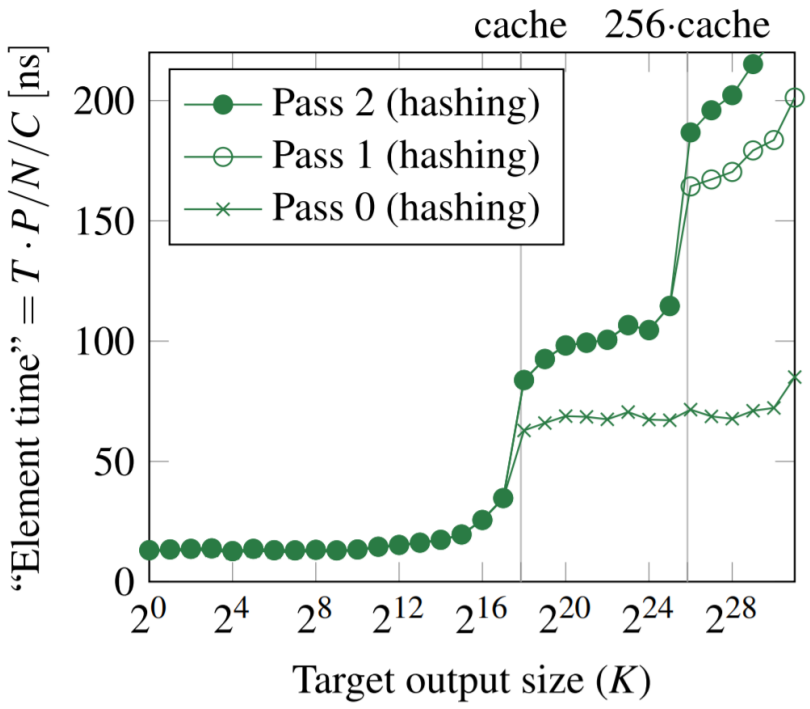
If the number of groups is smaller than the cache, HASHINGONLY computes the entire result in cache. Otherwise, it recurses until it can. PARTITIONALWAYS does not– doesn't know the right depth to recurse to before hashing pass.

2

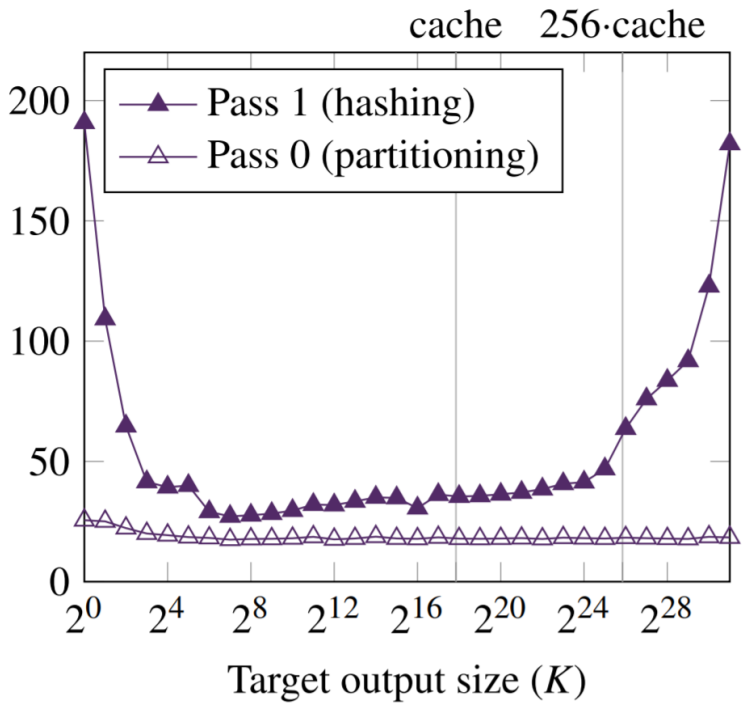
If K is much bigger than cache, partitioning is much faster. Hashing suffers from non-sequential memory accesses and wasted space. Tuning from before helps partition achieve high throughput.

3

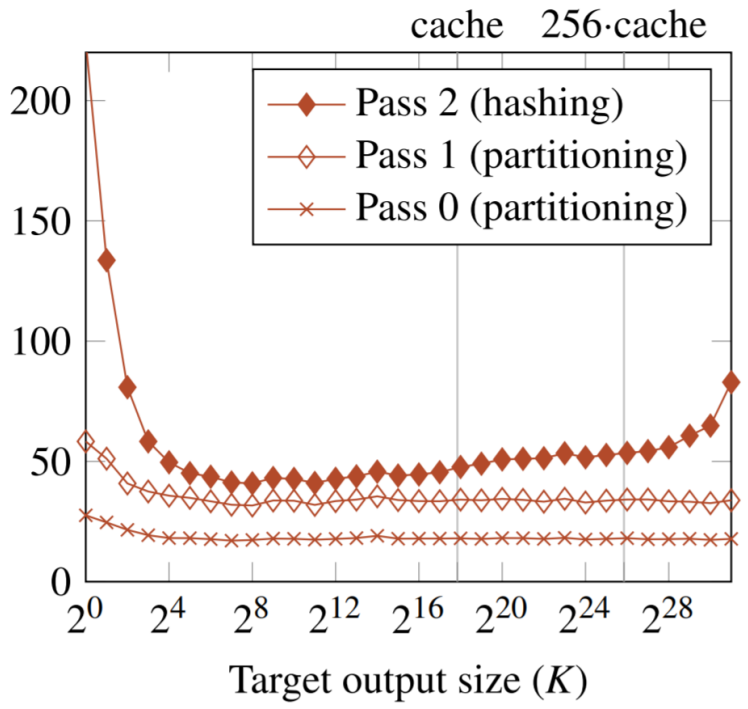
If data uniform, use partition until number of groups per partition is small, and then hash. If data clustered, hashing can reduce size significantly (even though more groups that fit in cache)– what to do?



(a) HASHINGONLY

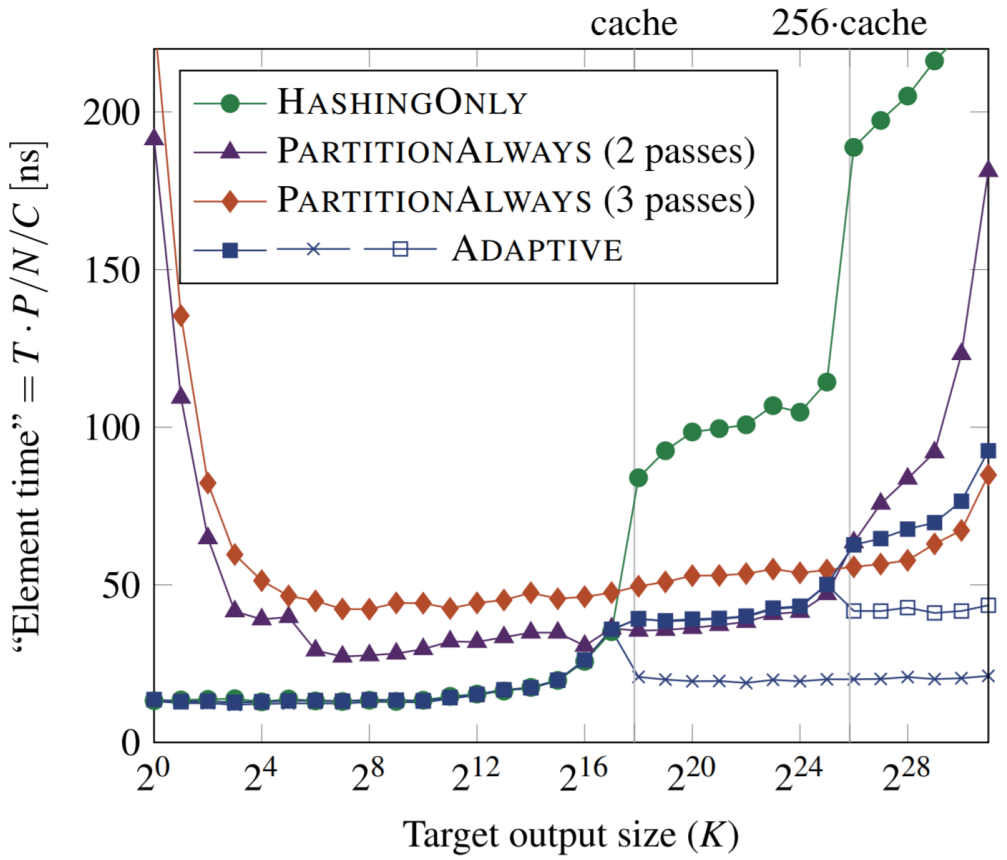


(b) PARTITIONALWAYS (2 passes)



(c) PARTITIONALWAYS (3 passes)

HASHING *IS* SORTING…
BUT WHEN TO PICK BETWEEN THEM?



Adaptive Method

1. Start with hashing
2. When hash table gets full, determine factor by which input has been reduced (number of input rows vs. size of hash table)

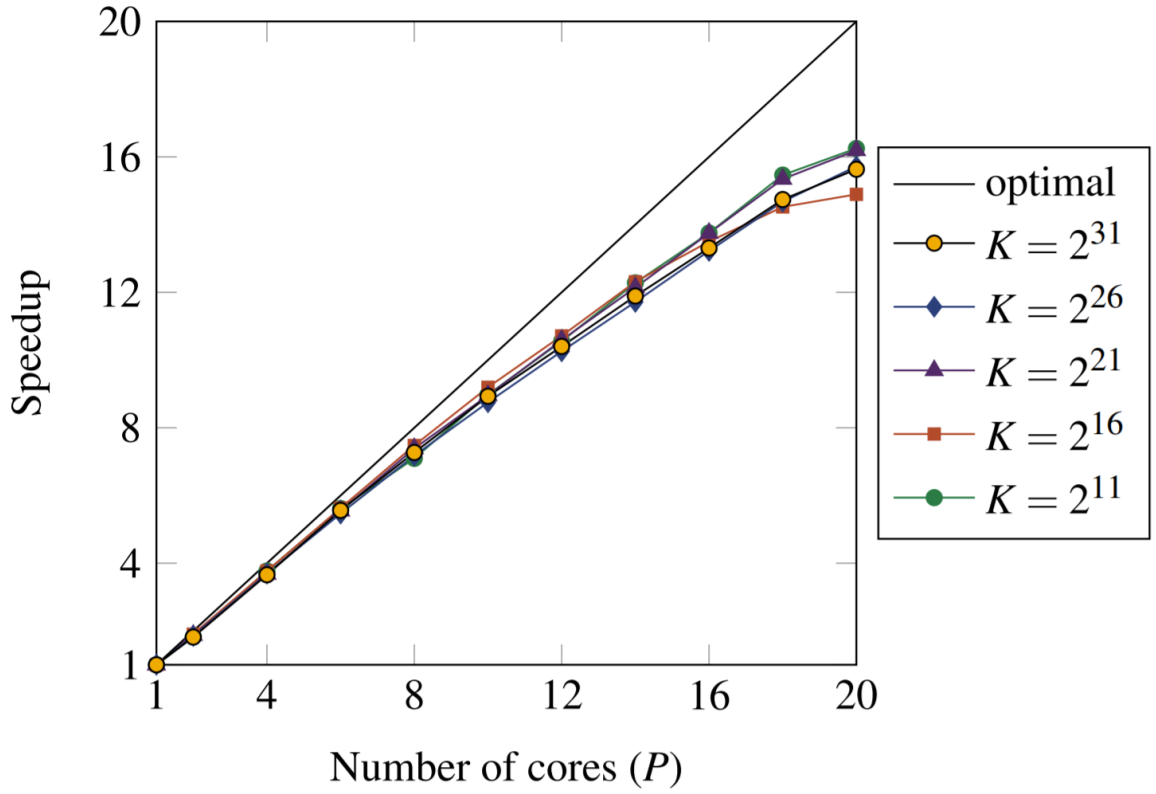
$$\alpha := \frac{n_{in}}{n_{out}}$$

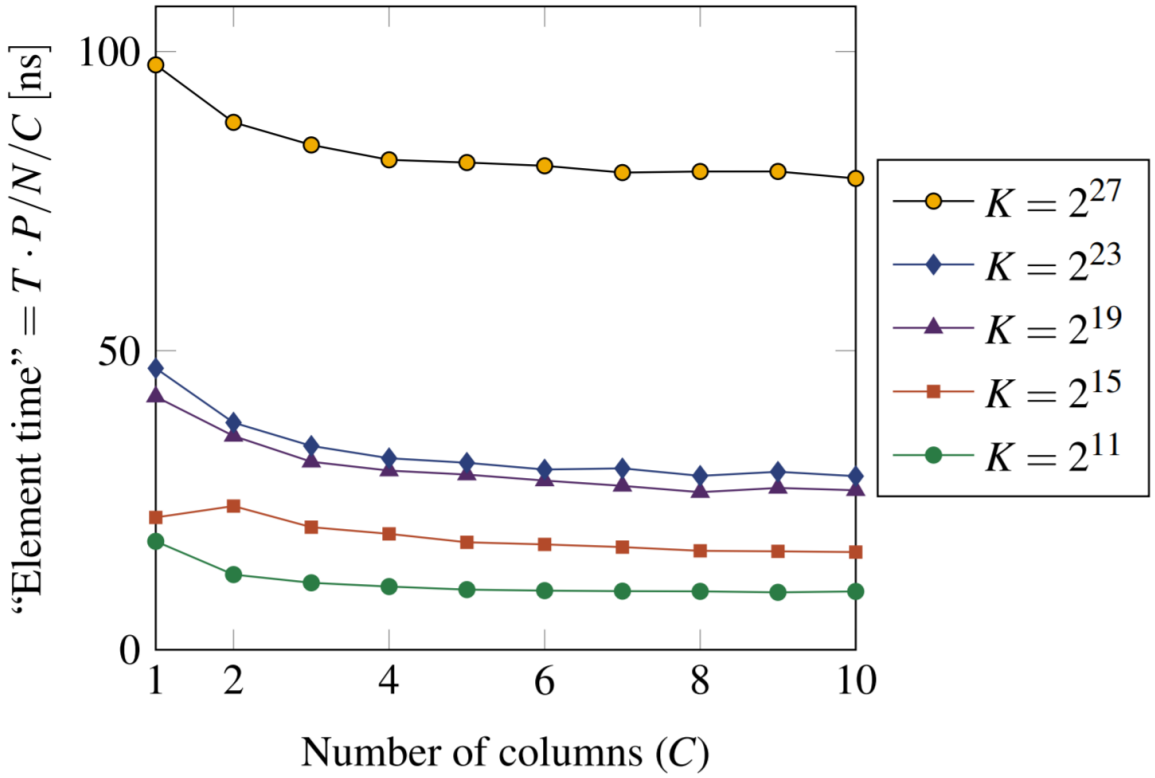
3. If $\alpha > \alpha_0$ for some threshold, switching to partitioning
4. When $n_{in} = c \cdot$ cache for some constant, algorithm switches back to hashing

AGENDA

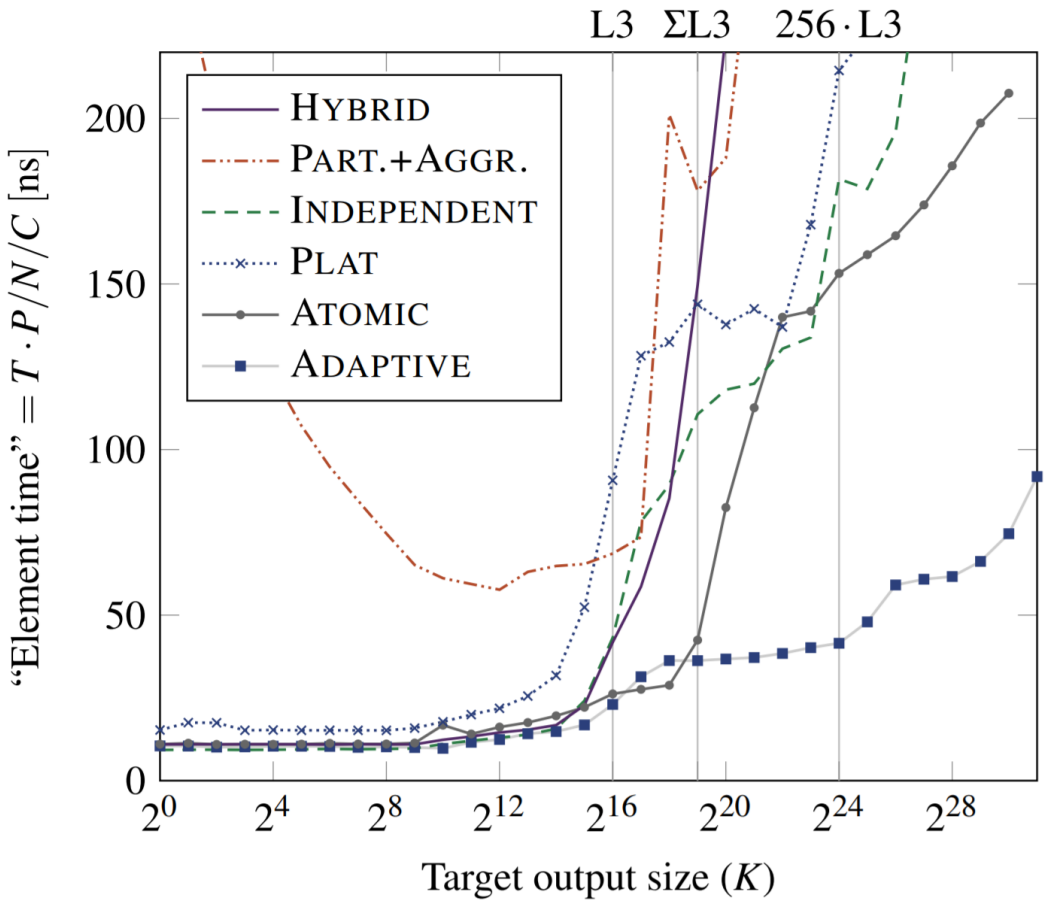
- Motivation
- Duality of hashing and sorting
- Efficient hashing and sorting primitives
- Design and analysis of aggregation algorithm
- Implementation and optimization of algorithm
- **Evaluation and benchmarking**
- Discussion

SCALABILITY
ON 2^{31} ROWS WITH 64-BIT INT COLUMNS





COMPARISON TO STATE-OF-THE-ART
TUNED TO L3-CACHE SIZE FOR 1-COL DB



Hybrid: thread aggregates its partition into private hash fixed to its part of shared L3 cache; LRU strategy for evictions.

Atomic: all threads on single, shared hash table.

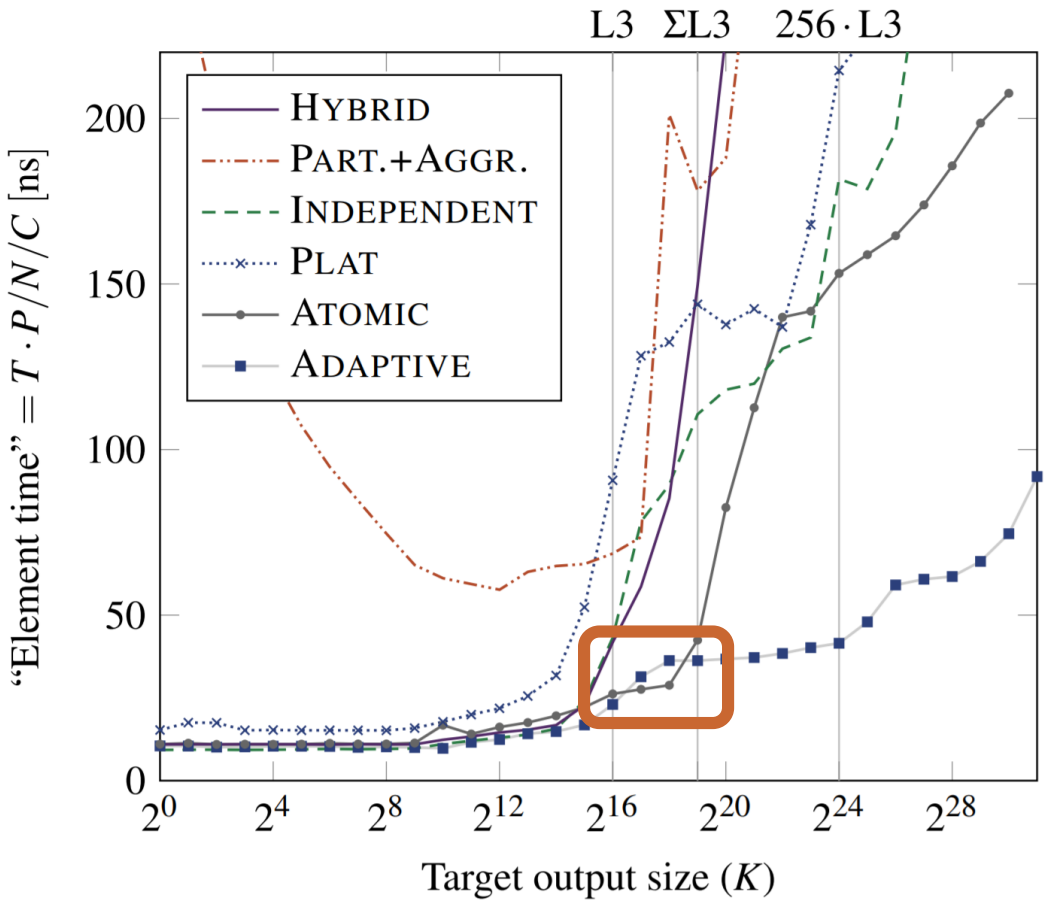
Independent: first pass for threads to produce hash table of its part of the input, which are then split and merged in parallel.

Partition-And-Aggregate: Partition entire input by hash value and then merge each partition into its part of a hash table.

PLAT: Each thread aggregates into a private fixed-size hash, and when full, entries overflown into hash partitions which are merged later.

Adaptive: this paper.

COMPARISON TO STATE-OF-THE-ART
TUNED TO L3-CACHE SIZE FOR 1-COL DB



Hybrid: thread aggregates its partition into private hash fixed to its part of shared L3 cache; LRU strategy for evictions.

Atomic: all threads on single, shared hash table.

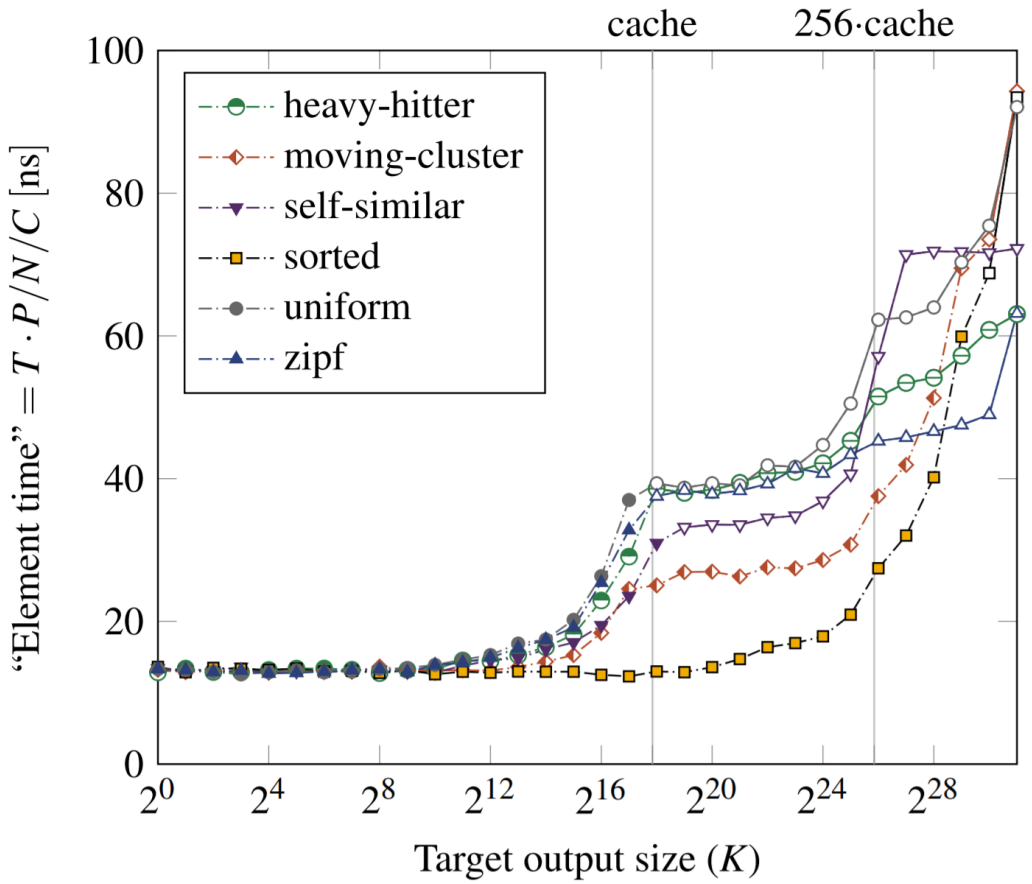
Independent: first pass for threads to produce hash table of its part of the input, which are then split and merged in parallel.

Partition-And-Aggregate: Partition entire input by hash value and then merge each partition into its part of a hash table.

PLAT: Each thread aggregates into a private fixed-size hash, and when full, entries overflown into hash partitions which are merged later.

Adaptive: this paper.

SKEW RESISTANCE
PERFORMANCE BY DISTRIBUTION TYPE



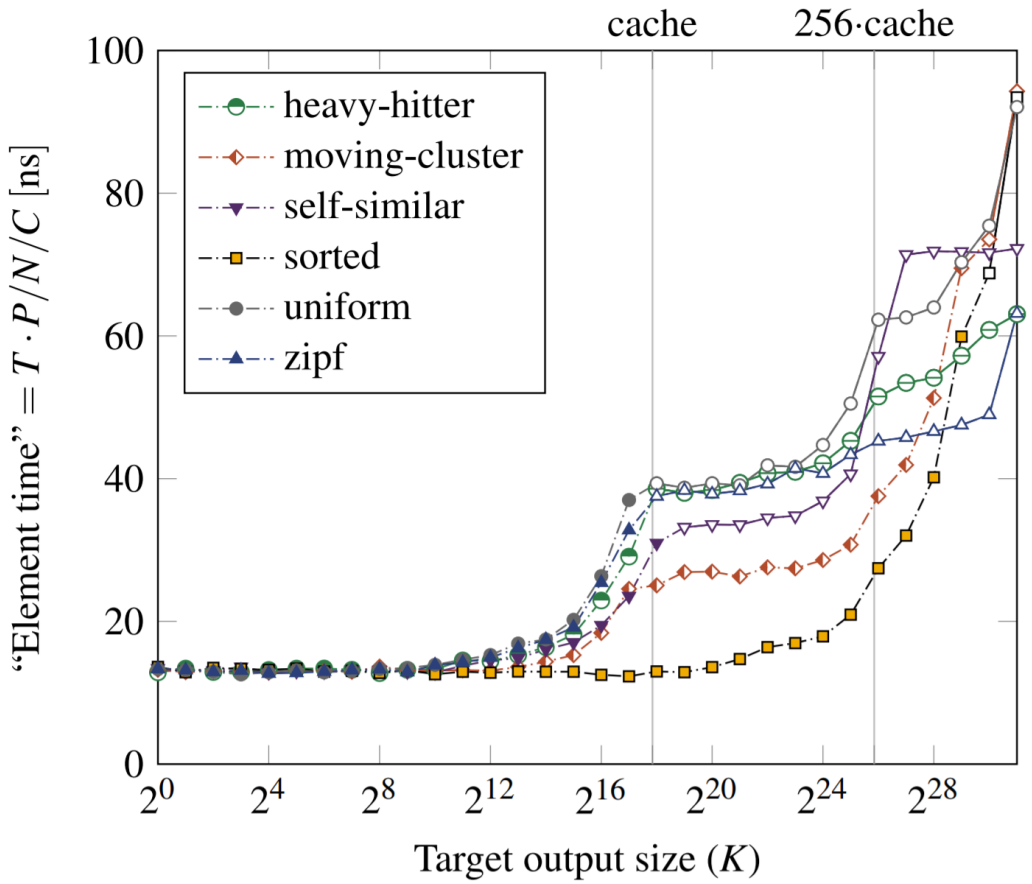
Heavy-hitter: 50% of records have the key 1, and the rest are uniformly distributed between 2 and K.

Moving cluster: keys chosen uniformly from sliding window of size 1024.

Self-similar: Pareto distribution with 80-20 proportion.

Zipfian: type of power-law distribution

SKEW RESISTANCE
PERFORMANCE BY DISTRIBUTION TYPE



Why does Adaptive perform better on skewed distributions?

Heavy-hitter: 50% of records have the key 1, and the rest are uniformly distributed between 2 and K.

Moving cluster: keys chosen uniformly from sliding window of size 1024.

Self-similar: Pareto distribution with 80-20 proportion.

Zipfian: type of power-law distribution

AGENDA

- Motivation
- Duality of hashing and sorting
- Efficient hashing and sorting primitives
- Design and analysis of aggregation algorithm
- Implementation and optimization of algorithm
- Evaluation and benchmarking
- **Discussion**

CONCLUSION

- Movement of data is fundamentally the limiting factor
- In external memory model, (optimized) sorting and (optimized) hashing are equivalent in terms of cache-line transfers
- Development of algorithmic framework that leverages both
- Tune routines to modern hardware
- Outperforms all competitors

DISCUSSION

- Bound on cache-line transfers we established: is this a bound on cache-line transfers for an aggregation query?
- Contradicts other work done on efficient `JOINS` where denser storage preferred: consensus on what works best?
- Thoughts on how to engineer other aggregation algorithms (e.g. Atomic) to be transparent to output size K?
- Where does this hashing/sorting tradeoff occur in other domains?
- Writing style: sequential and logically laid out, but self-aggrandizing; use of phrases like "time gracefully decays"