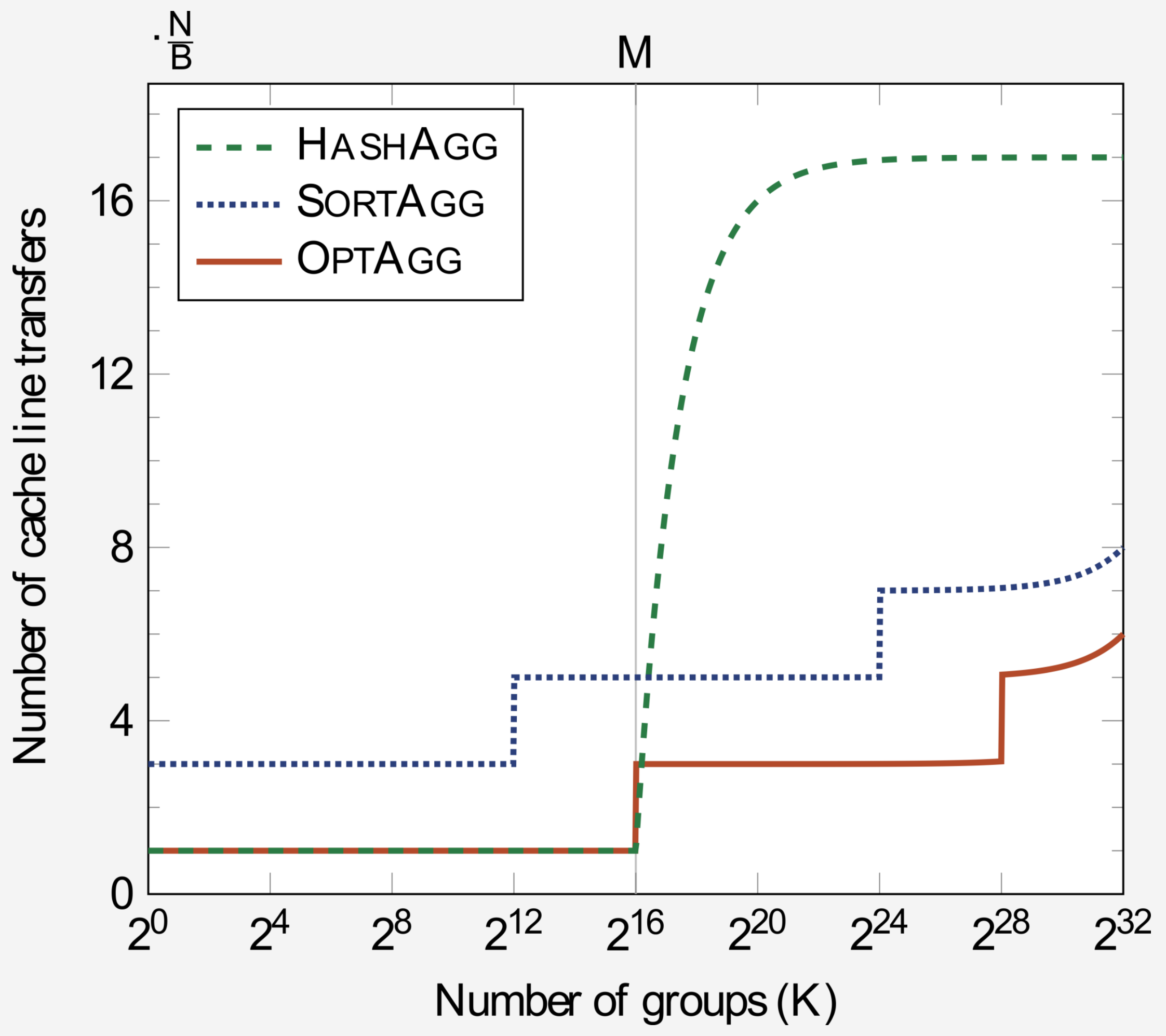
Cache-Efficient Aggregation: Hashing *Is* Sorting

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

SIGMOD, June 3, 2015

1. Textbook aggregation algorithms

- **Hash-Aggregation**: Insert every row into hash map with grouping attributes as key and aggregate to existing intermediate result.
  → In-cache processing of **small number of groups**.
- **Sort-Aggregation**: Sort input by grouping attributes, then aggregate consecutive rows in a single pass.
  → Efficient external sort for **large number of groups**.



M = cache size
B = block size
N = input size
K = output size

- Traditional approach: **Optimizer selects** physical operator based on cardinality estimation → error prone.

2. Our approach: Hashing and Sorting mixed in a single operator

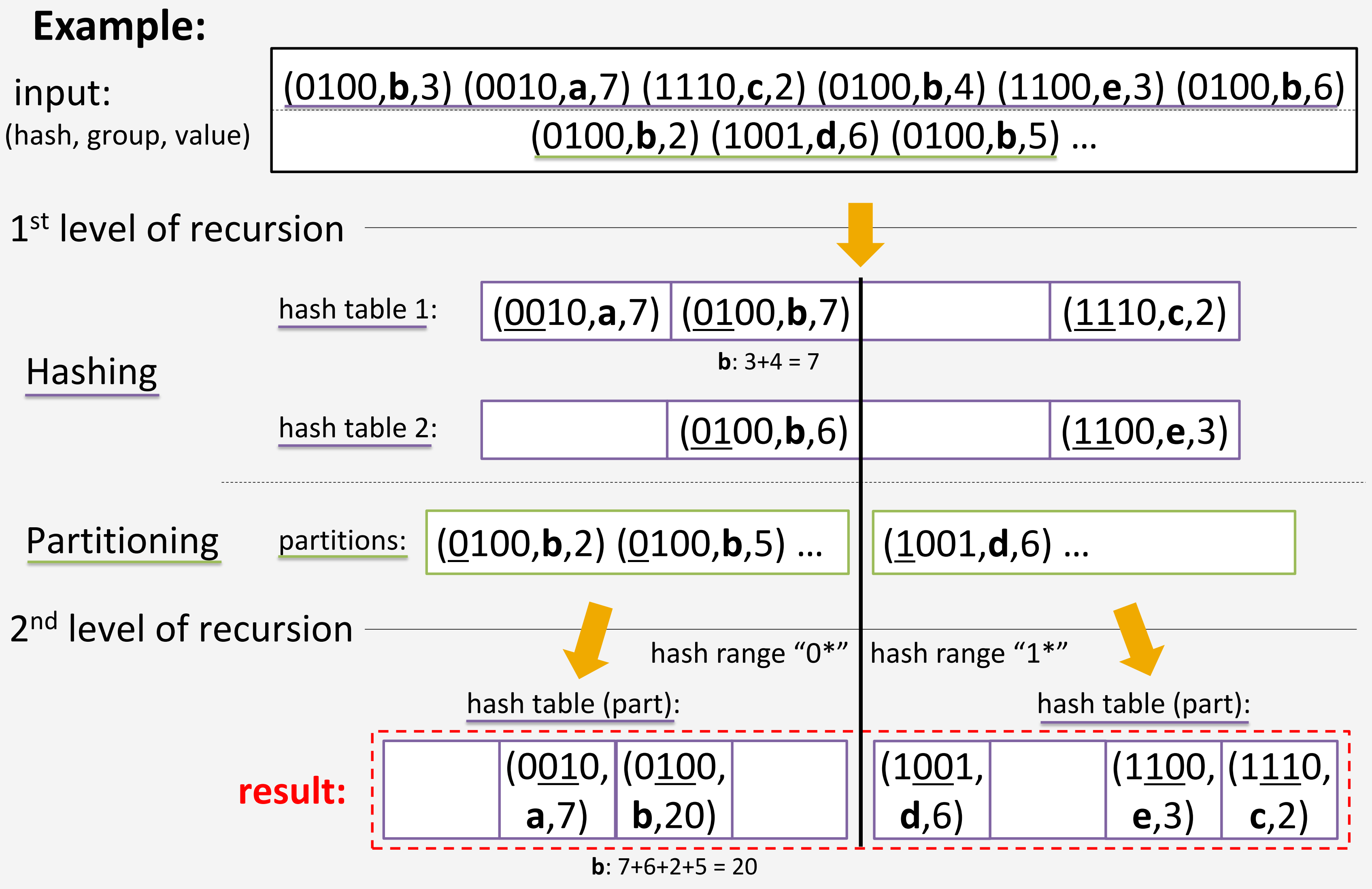
Key observation: Hashing is the same as *Sorting by hash value!*

Idea: design an aggregation operator like a Divide'n'Conquer sort algorithm on the hash values of the grouping attributes.

Use two subroutines in each level of recursion:

- "**Hashing**": insert (and aggregate) into *series* of hash tables, each of cache size → efficient (sort of).
- "**Partitioning**": append (w/o aggregation) to hash-partitions (like radix sort) → only sequential access → efficient.

Example:


input:
(hash, group, value)
(0100,**b**,3) (0010,**a**,7) (1110,**c**,2) (0100,**b**,4) (1100,**e**,3) (0100,**b**,6) (0100,**b**,2) (1001,**d**,6) (0100,**b**,5) …

1st level of recursion

Hashing
hash table 1: (0010,**a**,7) (0100,**b**,7) (1110,**c**,2)
b: 3+4 = 7
hash table 2: (0100,**b**,6) (1100,**e**,3)

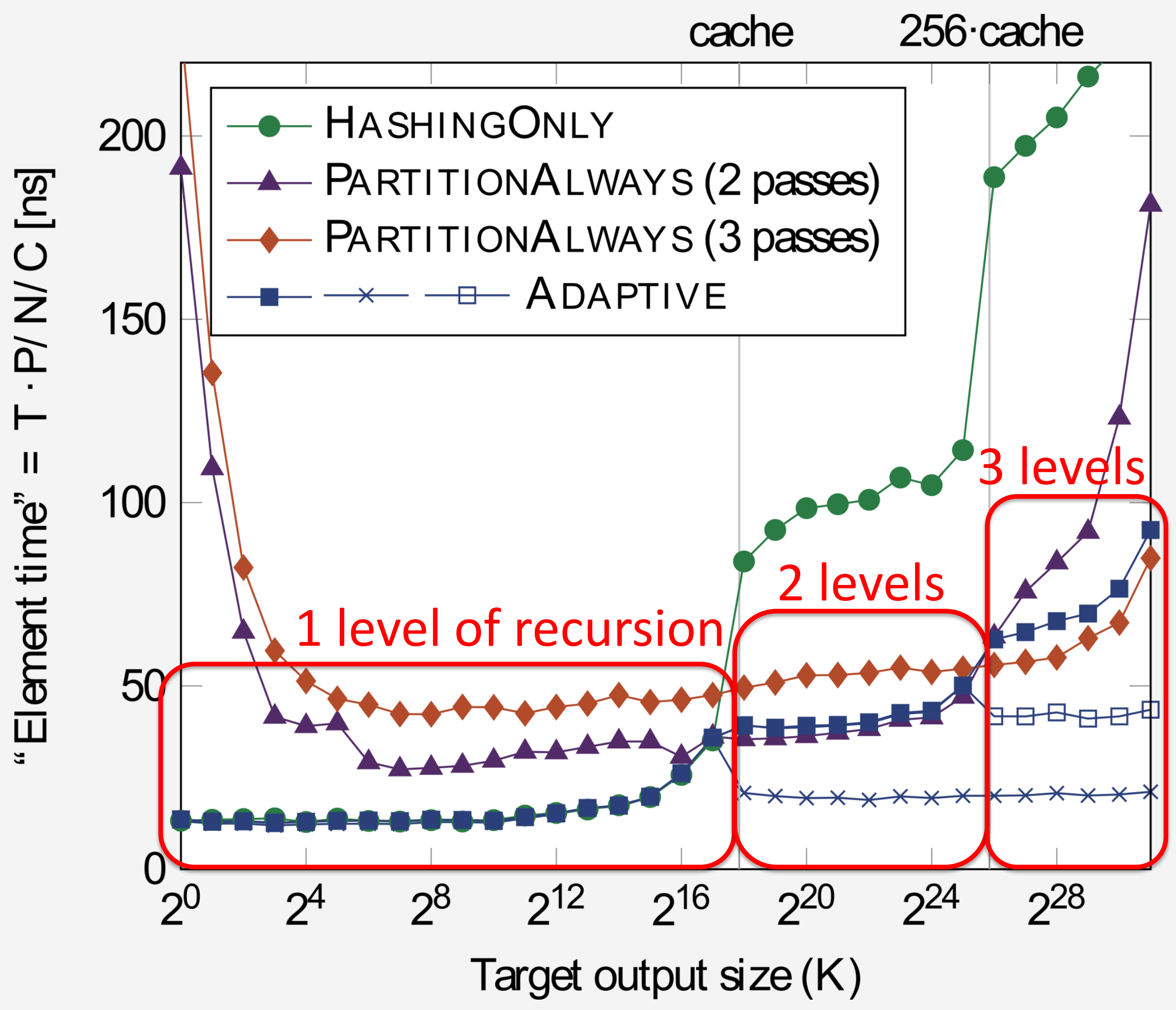
Partitioning
partitions: (0100,**b**,2) (0100,**b**,5) … | (1001,**d**,6) …

2nd level of recursion
hash range "0*" | hash range "1*"
hash table (part): | hash table (part):
result: (0010,**a**,7) (0100,**b**,20) | (1001,**d**,6) (1100,**e**,3) (1110,**c**,2)
b: 7+6+2+5 = 20

- The two routines produce a **mix of hash tables and partitions**.
- Some groups may still occur several times after the first pass → we **recurse into hash ranges of all intermediate results combined** until every (sub)range of hash values is fully aggregated.
- Next question: ***when** to use **which** routine*?

3. Our adaptation mechanism

- Start with **Hashing** until hash table full.
- **If Hashing was "worth it"**, i.e., if the input was aggregated "enough", thus reducing the amount of work for recursive processing, do **Hashing again**.
- **Otherwise do Partitioning** for "some time", then start over.
- The paper gives quantifications for "enough" and "some time".



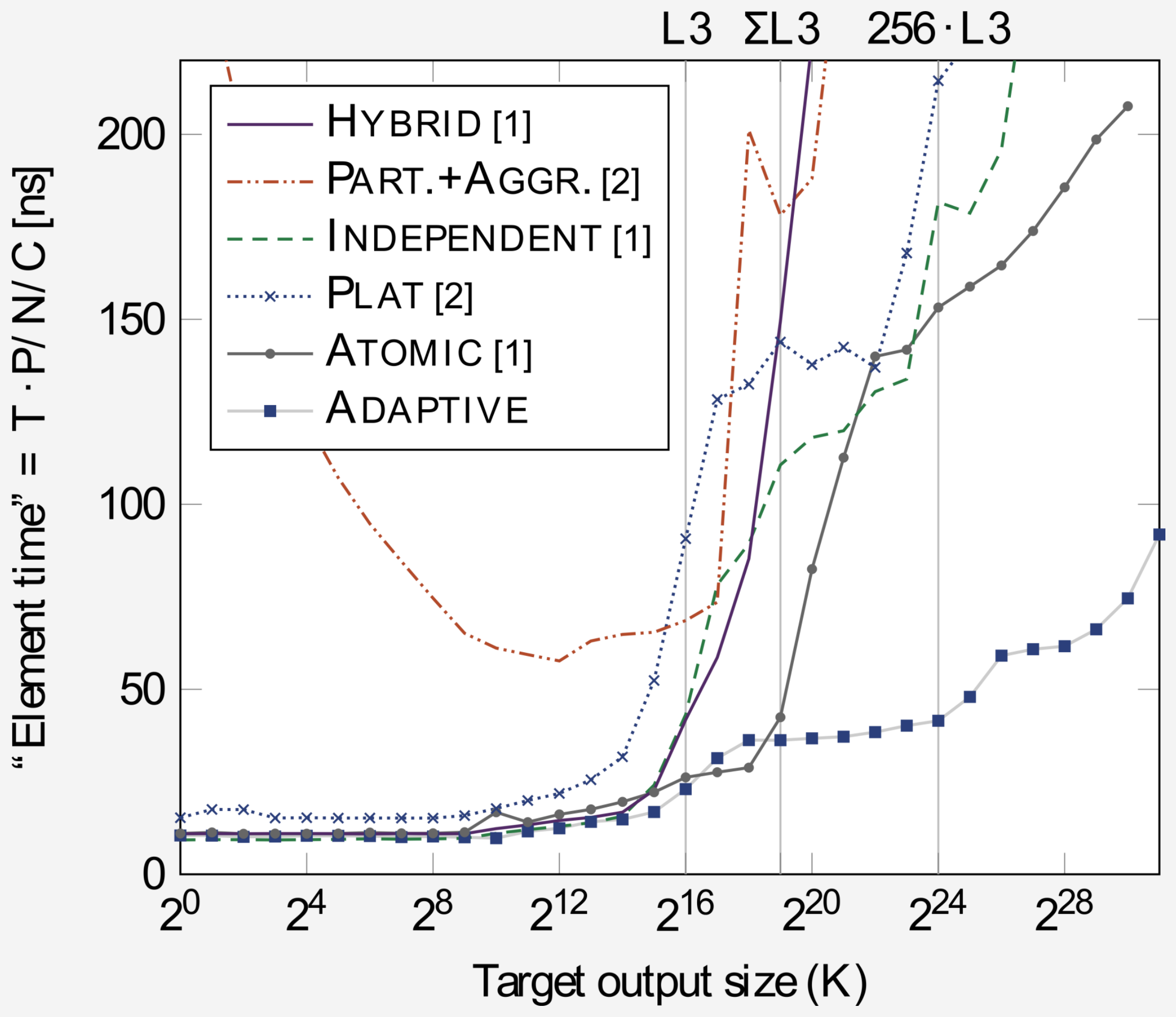
2 Xeon E7-8870 CPUs
(each 10 cores),
uniform distribution.

T = runtime
P = #cores (20)
N = #input rows (2^{32})
C = #columns (1)

Without prior information, this mechanism **adapts to the data** by:

- ending recursion with in-cache hashing **as early as possible**,
- using the extremely fast partition routine (97% of the speed of `memcpy`) **as long as necessary**.

4. Evaluation: Comparison with prior work



2 Xeon E7-8870 CPUs
(each 10 cores),
uniform distribution.

T = runtime
P = #cores (20)
N = #input rows (2^{32})
C = #columns (1)

Result:

- Our algorithm ("Adaptive") **faster** than all others [1,2] for K > 2^{20}.
- Up to **factor 3.7 speedup** to second best.

[1] John Cieslewicz, Kenneth A. Ross. Adaptive Aggregation on Chip Multiprocessors. In *PVLDB*, 2007.

[2] Yang Ye, Kenneth A. Ross, Norases Vesdapunt. Scalable Aggregation on Multicore Processors. In *Proc. of DaMoN*, 2011.

5. Outlook

What else to expect in the paper?

- How to **parallelize**?
- How to integrate with **JiT** and **column-wise processing**?
- How to tune hashing and sorting to **modern hardware**?
- How to determine **thresholds**?
- Why does it also work well in presence of **skew**?

SAP HANA
Database Campus