Cache-Efficient Aggregation: Hashing *Is* Sorting

Ingo Müller[1†‡], Peter Sanders[2†], Arnaud Lacurie[3‡]
Wolfgang Lehner[4*], Franz Färber[5‡]

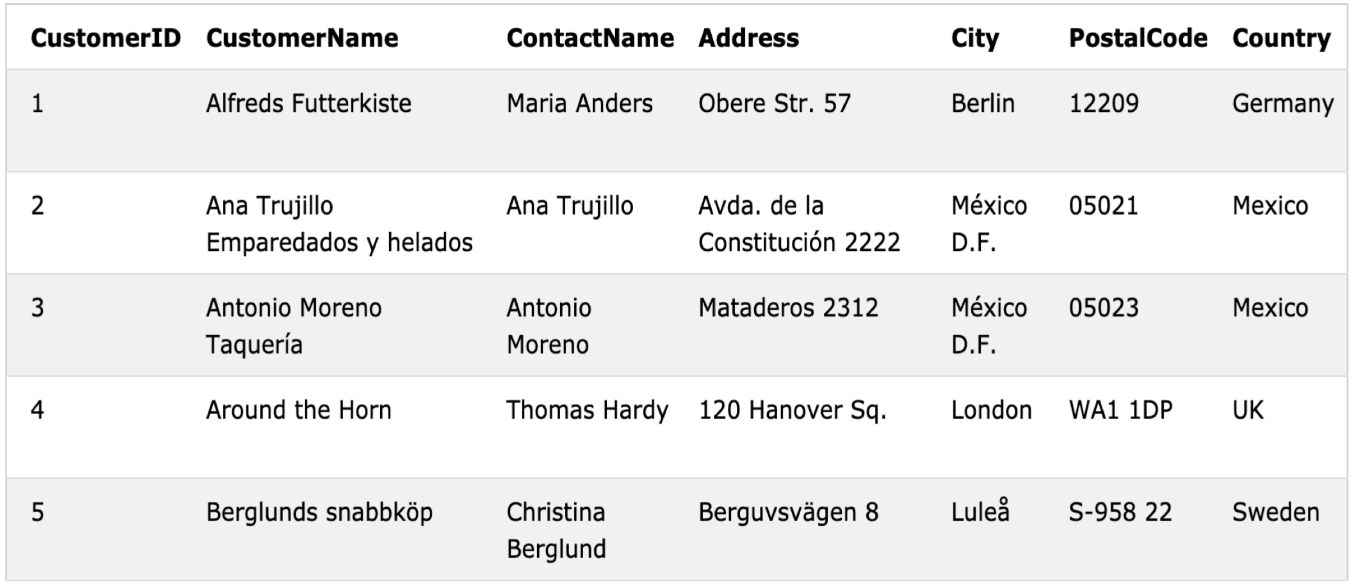
[†]Karlsruhe Institute of Technology, [‡]SAP SE, [*]Dresden University of Technology

[1]ingo.mueller@kit.edu, [2]sanders@kit.edu, [3]arnaud.lacurie@sap.com,
[4]wolfgang.lehner@tu-dresden.de, [5]franz.faerber@sap.com

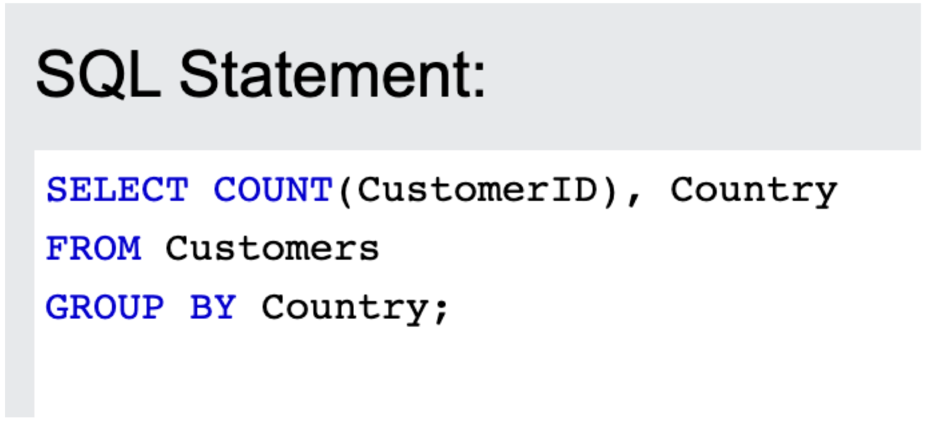
Presenter: Tao Sun

Background

GROUPING with AGGREGATION is one of the most expensive relational database operators. The dominant cost of AGGREGATION is, as with most relational operators, the movement of the data – from main memory to cache.

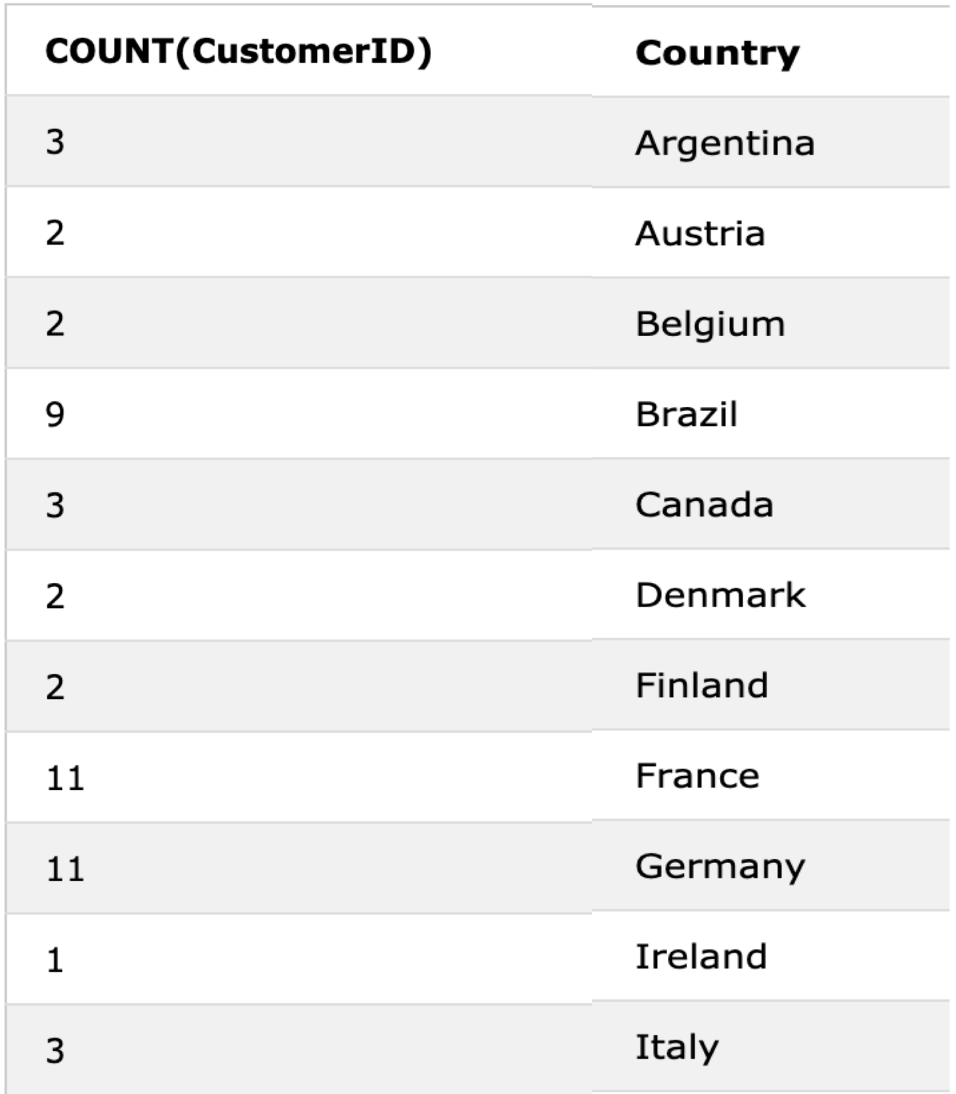


CustomerID	CustomerName	ContactName	Address	City	PostalCode	Country
1	Alfreds Futterkiste	Maria Anders	Obere Str. 57	Berlin	12209	Germany
2	Ana Trujillo Emparedados y helados	Ana Trujillo	Avda. de la Constitución 2222	México D.F.	05021	Mexico
3	Antonio Moreno Taquería	Antonio Moreno	Mataderos 2312	México D.F.	05023	Mexico
4	Around the Horn	Thomas Hardy	120 Hanover Sq.	London	WA1 1DP	UK
5	Berglunds snabbköp	Christina Berglund	Berguvsvägen 8	Luleå	S-958 22	Sweden



SQL Statement:

```sql
SELECT COUNT(CustomerID), Country
FROM Customers
GROUP BY Country;
```

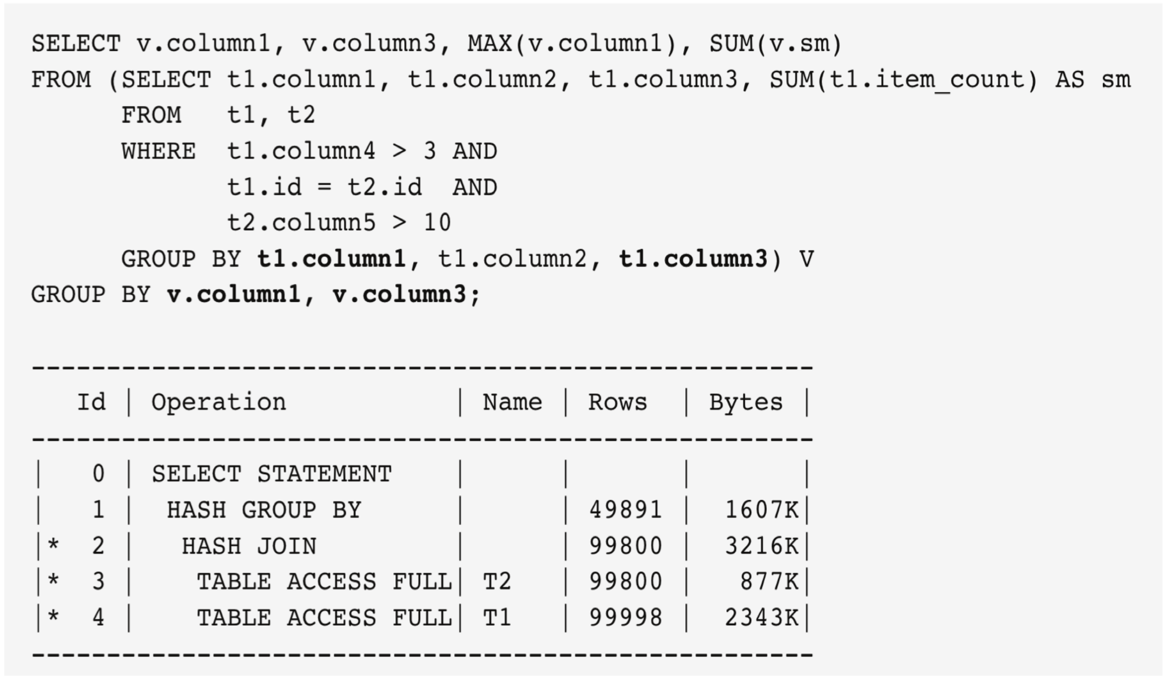


COUNT(CustomerID)	Country
3	Argentina
2	Austria
2	Belgium
9	Brazil
3	Canada
2	Denmark
2	Finland
11	France
11	Germany
1	Ireland
3	Italy

Hashing and Sorting for aggregation

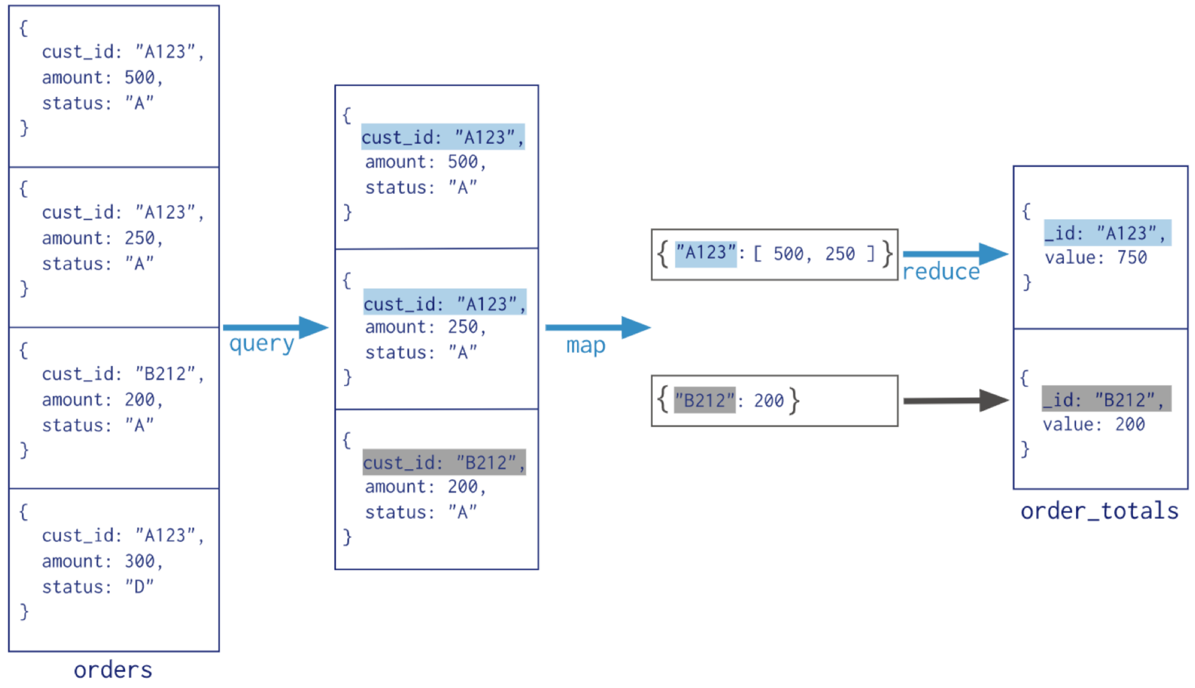
HASHAGGREGATION inserts the input rows into a hash table, using the grouping attributes as key and aggregating the remaining attributes in-place.

SORTAGGREGATION first sorts the rows by the grouping attributes and then aggregates the consecutive rows of each group.



```
SELECT v.column1, v.column3, MAX(v.column1), SUM(v.sm)
FROM (SELECT t1.column1, t1.column2, t1.column3, SUM(t1.item_count) AS sm
      FROM   t1, t2
      WHERE  t1.column4 > 3 AND
             t1.id = t2.id   AND
             t2.column5 > 10
      GROUP BY t1.column1, t1.column2, t1.column3) V
GROUP BY v.column1, v.column3;

----------------------------------------------------
   Id | Operation            | Name | Rows   | Bytes |
----------------------------------------------------
|   0 | SELECT STATEMENT     |      |        |       |
|   1 |  HASH GROUP BY       |      | 49891  |  1607K|
|*  2 |   HASH JOIN          |      | 99800  |  3216K|
|*  3 |    TABLE ACCESS FULL| T2   | 99800  |   877K|
|*  4 |    TABLE ACCESS FULL| T1   | 99998  |  2343K|
----------------------------------------------------
```



{ cust_id: "A123", amount: 500, status: "A" }
{ cust_id: "A123", amount: 250, status: "A" }
{ cust_id: "B212", amount: 200, status: "A" }
{ cust_id: "A123", amount: 300, status: "D" }
orders
query
{ cust_id: "A123", amount: 500, status: "A" }
{ cust_id: "A123", amount: 250, status: "A" }
{ cust_id: "B212", amount: 200, status: "A" }
map
{ "A123": [500, 250] }
{ "B212": 200 }
reduce
{ _id: "A123", value: 750 }
{ _id: "B212", value: 200 }
order_totals

The consensus is that HASHAGGREGATION is better if the number of groups is small enough such that the output fits into the cache, and SORTAGGREGATION is better if the number of groups is very large.

Paper Contribution

We obtain a novel relational aggregation algorithm that has very low constant factors on modern hardware. It is cache-efficient, highly parallelizable on modern multi-core systems, and operating at a speed close to the memory bandwidth.

- The two approaches have exactly the same costs in terms of cache line transfers.
- Design an algorithmic framework that allows seamless switching between hashing and sorting during execution.
- Show how to achieve very low constant factors for both the hashing and the sorting routine by tuning them to modern hardware.

Sort-based aggregation

External memory model

$$N = \text{number of input rows}$$
$$K = \text{number of groups in the input}$$
$$M = \text{number of rows fitting into cache}$$
$$B = \text{number of rows per single cache line}$$

Bucket sort is used

- there are as many leaves in the call tree as there are cache lines in the input: N/B .
- The tree has degree M/B since the number of partitions is limited by the number of buffers that fit into cache.
- If we assume that the tree is somewhat balanced, it has a height of $\left\lceil \log_{\frac{M}{B}} \frac{N}{B} \right\rceil$.

Overall cost of SORTAGGREGATION

$$\textsc{SortAggStat}(N,K) = \overset{\text{Input Read and write}}{2 \cdot \frac{N}{B}} \cdot \left\lceil \log_{\frac{M}{B}} \frac{N}{B} \right\rceil + \overset{\text{Aggregation Input read}}{\frac{N}{B}} + \overset{\text{Aggregation Output Write}}{\frac{K}{B}}$$

When K < N. the recursion actually stops earlier than for the case where K = N. In fact, the call tree only has min(N/B ;K) leaves, at most one for each partition,

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

we merge the last bucket sort pass with the final aggregation pass. This completely eliminates one pass over the entire data.

$$\boxed{\textsc{SortAggOpt}(N,K) = \frac{N}{B} + 2 \cdot \frac{N}{B} \left(\left\lceil \log_{\frac{M}{B}} \frac{K}{B} \right\rceil - 1 \right) + \frac{K}{B}}$$

Hash-based aggregation

If K< M, the algorithm just needs the N/B cache line transfers for reading the input. If K > M, even only a fraction of M/K rows can be in the cache. The overall number of cache line transfers is therefore:

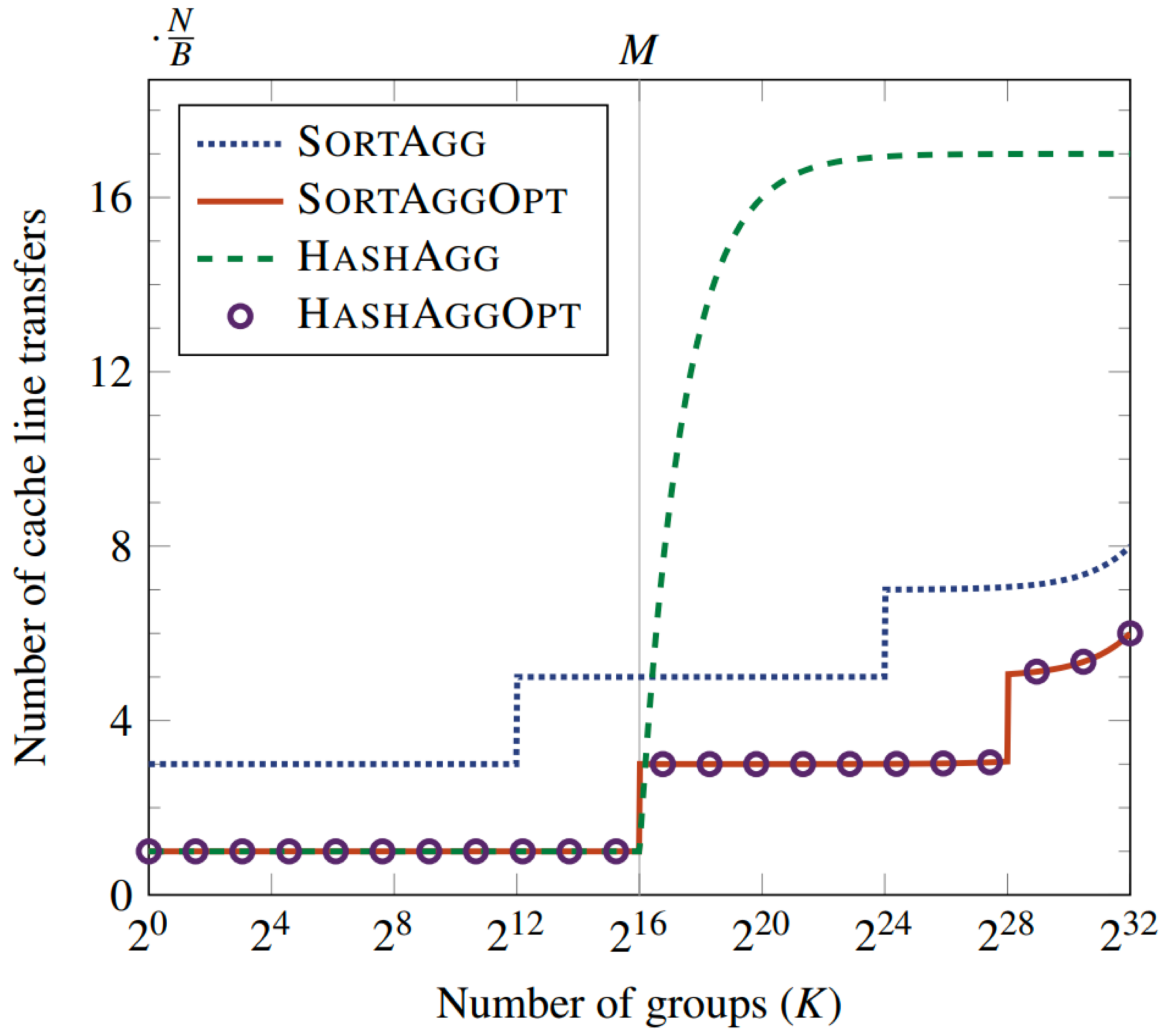
$$\textsc{HashAgg}(N,K) = \frac{N}{B} + \begin{cases} \frac{K}{B} & \text{if } K < M \\ 2 \cdot \left(1 - \frac{M}{K}\right) \cdot N & \text{otherwise} \end{cases}$$

As long as the output K is small enough to fit into the cache, HASHAGGREGATION is really fast. However as soon as the cache cannot hold the output anymore, HASHAGGREGATION triggers a cache miss for almost every input row, so the number of cache line transfers explodes

A common optimization to overcome this problem is to (recursively) partition the input by hash value and to apply HASHAGGREGATION on each partition separately. However the partitioning also entails costs, which are the same as the partitioning of bucket sort.

$$\boxed{\textsc{HashAggOpt}(N,K) = 2 \cdot \frac{N}{B} \left(\left\lceil \log_{\frac{M}{B}} \frac{K}{B} \right\rceil - 1 \right) + \frac{N}{B} + \frac{K}{B}}$$

Hashing and Sorting Comparison

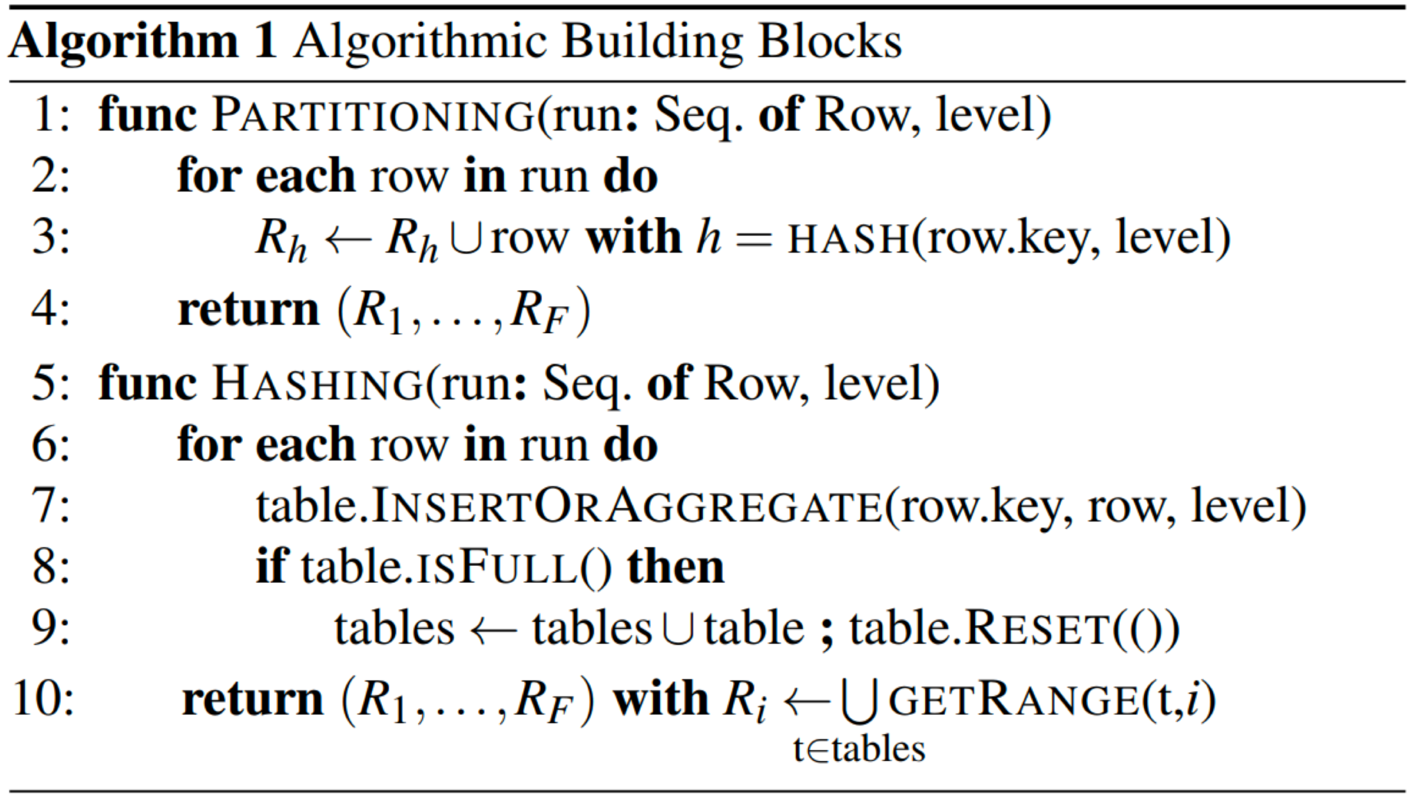


Mixing Hashing and Sorting

We can define the following two partitioning routines,

- plain partitioning by hash value called PARTITIONING (Line 1)
- a partitioning routine based on the creation of hash tables called HASHING (Line 5).

Both routines produce partitions in form of "runs"



Algorithm 1 Algorithmic Building Blocks

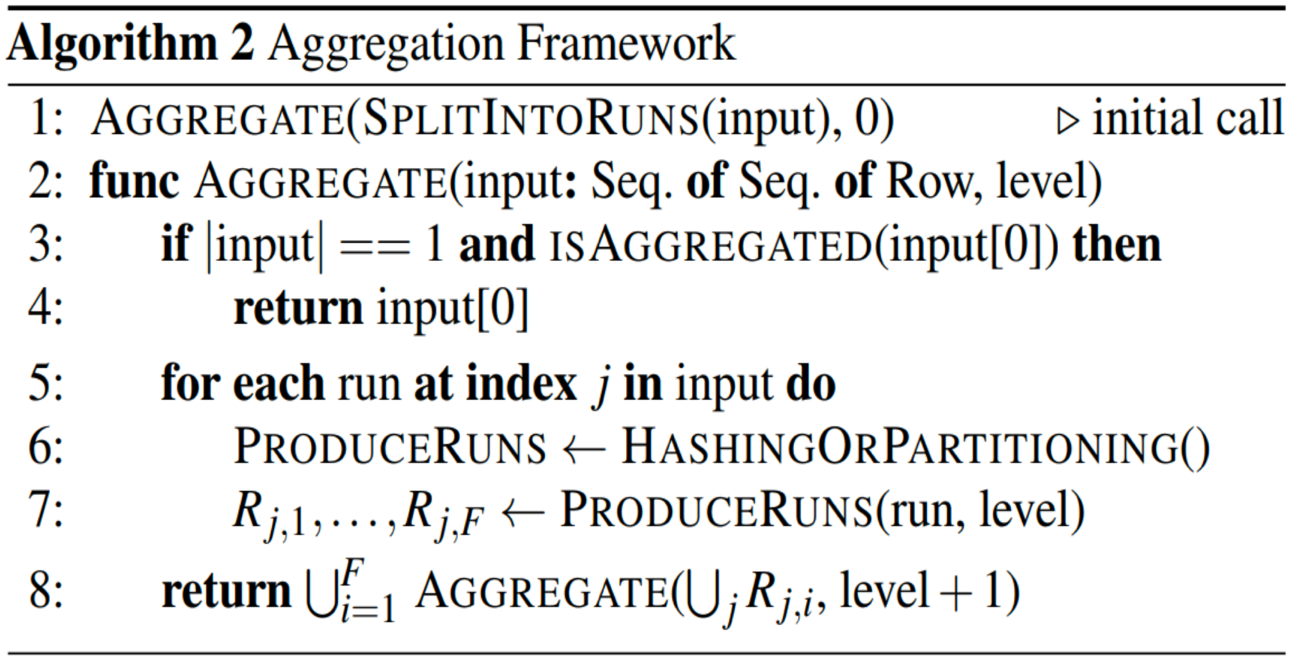
```
1:  func PARTITIONING(run: Seq. of Row, level)
2:      for each row in run do
3:          $R_h \leftarrow R_h \cup$ row with $h =$ HASH(row.key, level)
4:      return $(R_1, \ldots, R_F)$
5:  func HASHING(run: Seq. of Row, level)
6:      for each row in run do
7:          table.INSERTORAGGREGATE(row.key,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

PARTITIONING produces one run per partition by moving every row to its respective run.

HASHING starts with a first hash table of the size of the cache and replaces its current hash table with a new one whenever it is full. Every full hash table is split into one run per partition

working set of both HASHING and PARTITIONING is strictly limited to the CPU cache.

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)
```

first split into runs.

each run of the input is processed by one of the two routines selected by HASHING OR PARTITIONING

Once the entire input has been processed, the algorithm treats all runs of the same partition as a single bucket and recurses into the buckets one after each other.

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                    Parallelable
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)    Parallelable
```

Synchronization

System integration

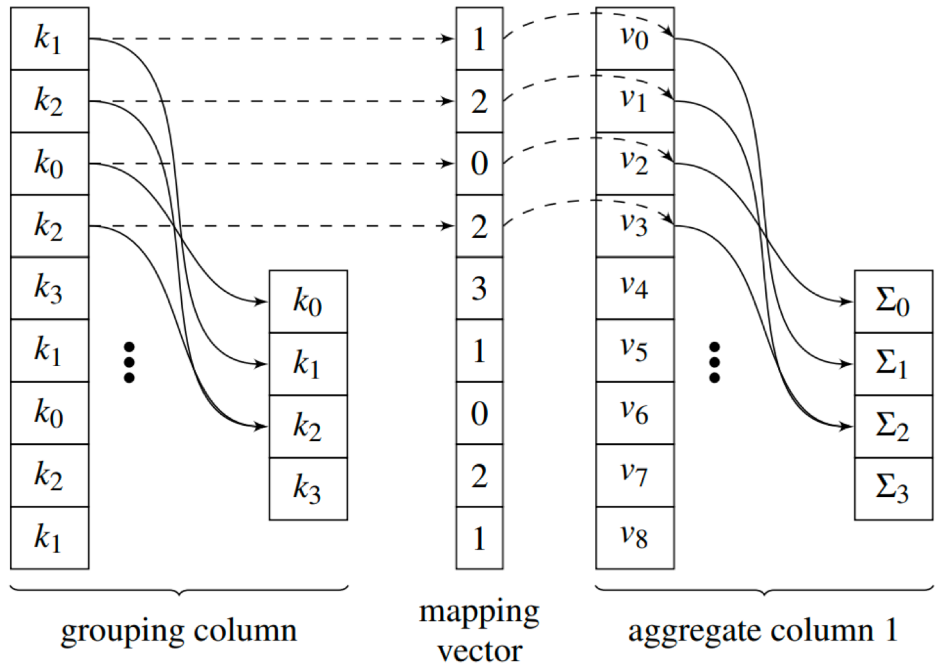


Figure 2: Column-wise processing.

With this approach, aggregation is split into two operators:

The first operator processes the grouping column and produces a vector with identifiers of the groups and a mapping vector, which maps every input row to the index of its group.

The second operator applies this mapping vector by aggregating every input value with the current aggregate of the group as indicated by the mapping vector and is executed once for each aggregate column

Disadvantages

- Require additional memory access to write and read the mapping vector.
- Ignores cache-efficient. For large outputs. there are often many more aggregate columns than grouping columns, this would even have a worse impact

Improvements

- Both HASHING and PARTITIONING produce a mapping vector but only for this run.
- This mapping is then applied to the corresponding parts of the aggregate columns.
- When the corresponding runs of all columns have been produced, the framework continues with the processing of the rest of the input.

Minimizing CPU costs

Hashing

We adapted the linear probing to work within blocks, such that we can cleanly split a table into ranges for the recursive calls. The final insertion costs of our implementation are below 6 ns per element. This is roughly 4 times more than an L1 cache access, but more than an order of magnitude faster than out-of-cache insertion,

Partitioning

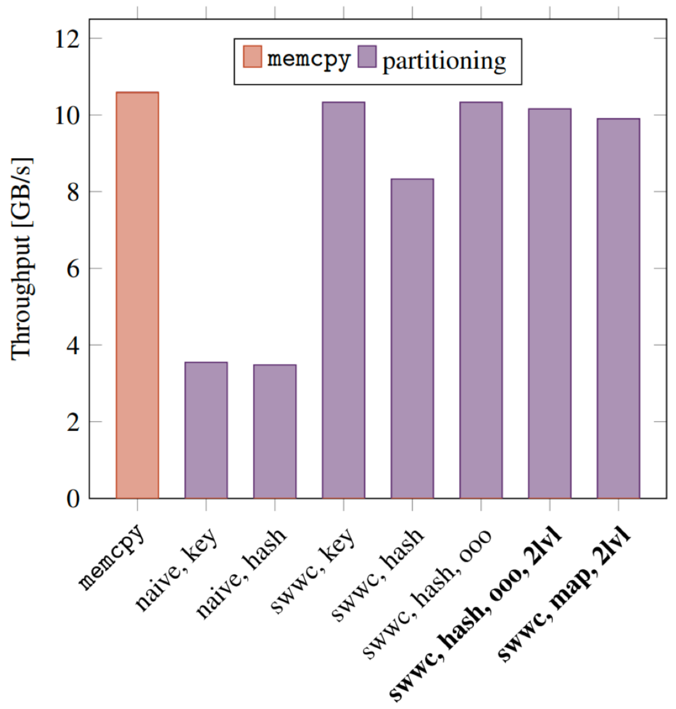


Figure 3: Microbenchmark of degenerated partitioning routines.

Software write-combining (swwc) is designed to avoid the read-before-write overhead. It consists in buffering one cache line per partition, which is flushed when it runs full using a non-temporal store instruction that by-passes the cache.

When to switch

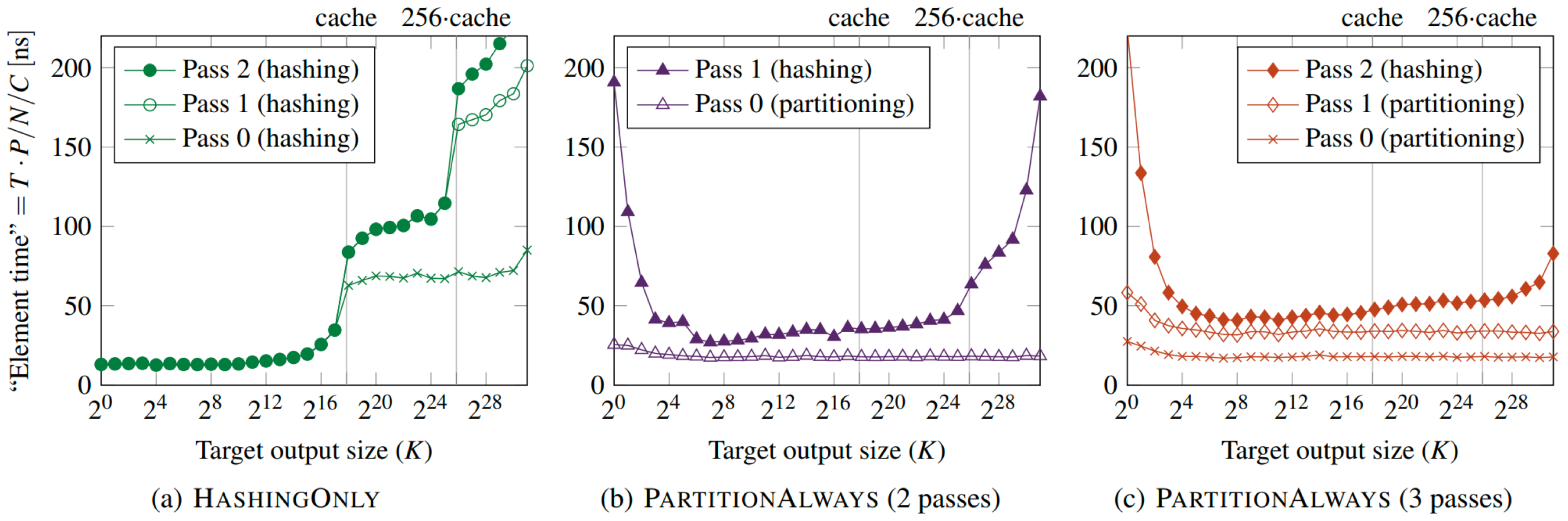


Figure 4: Breakdown of passes of illustrative aggregation strategies using $P = 20$ threads.

- HASHINGONLY automatically does the right number of passes: If K < cache, it computes the result in cache.
- PARTITIONING is much faster than HASHING if K > cache

Use PARTITIONING until the number of groups per partition is small enough such that HASHING can do the rest of the work in cache.

Adaptive switching strategy

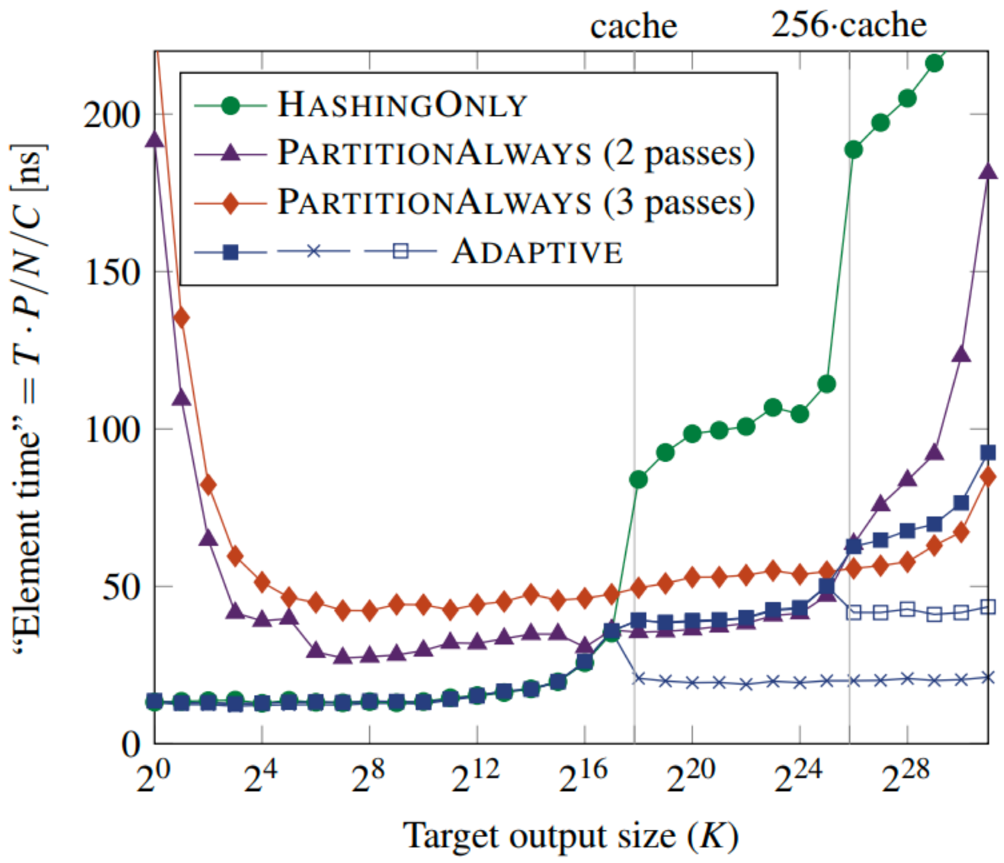


Figure 5: Adaptive strategy in comparison with HashingOnly and PartitionAlways (2 and 3 passes) using $P = 20$ threads.

The algorithm starts with HASHING. When a hash table gets full, the algorithm determines the factor a = n_in/ n_out, where n_in is the number of processed rows and n_out the size of the hash table.

If a > a0 for some threshold a0, HASHING was the better choice as the input was reduced significantly, so the algorithm continues with HASHING. Otherwise it switches to PARTITIONING.

Scalability

Number of cores

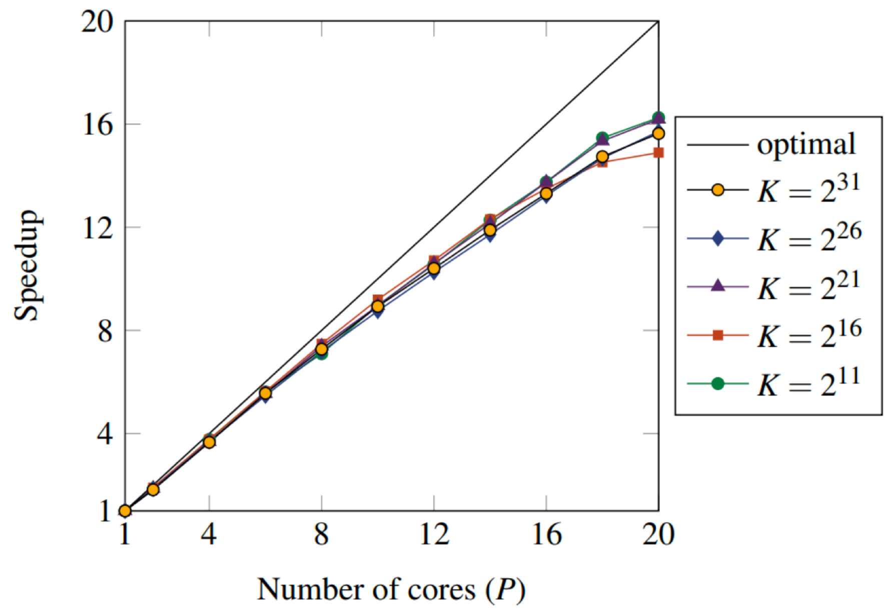


Figure 6: Speedup of ADAPTIVE compared to single core performance.

Figure 6 shows the speedup of ADAPTIVE for different numbers of groups K compared to its respective performance on a single core. As the plot shows, the speedup is around 16 on our 20 CPU cores no matter K, which is as close to optimal speedup as practical implementations usually get.

Number of columns

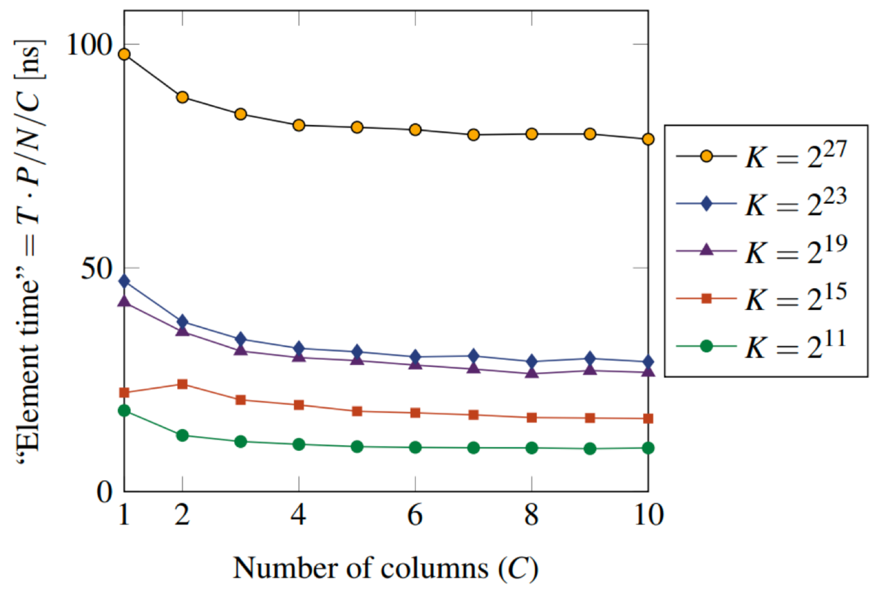
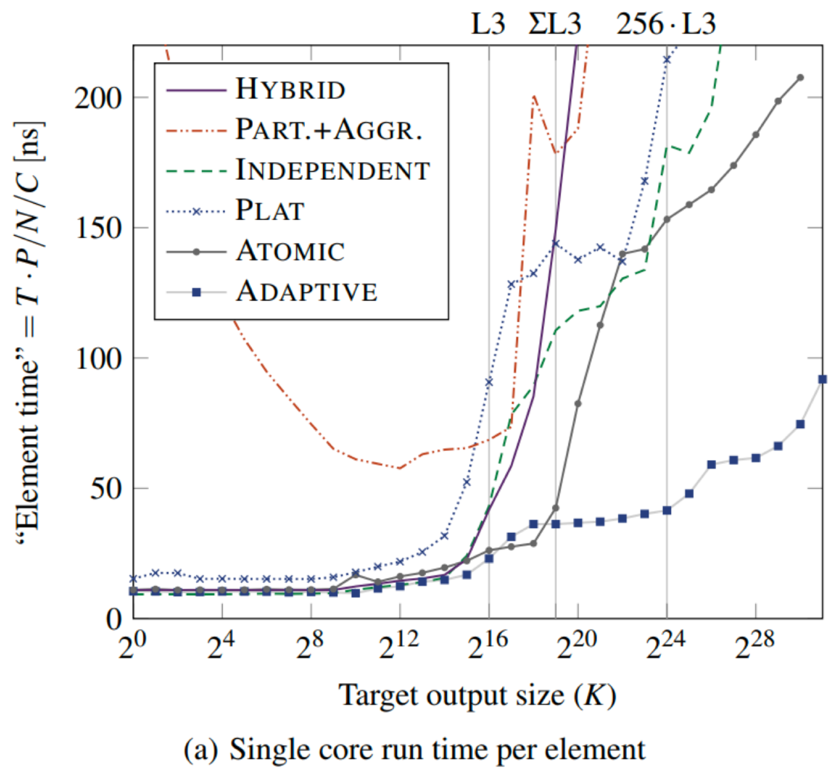


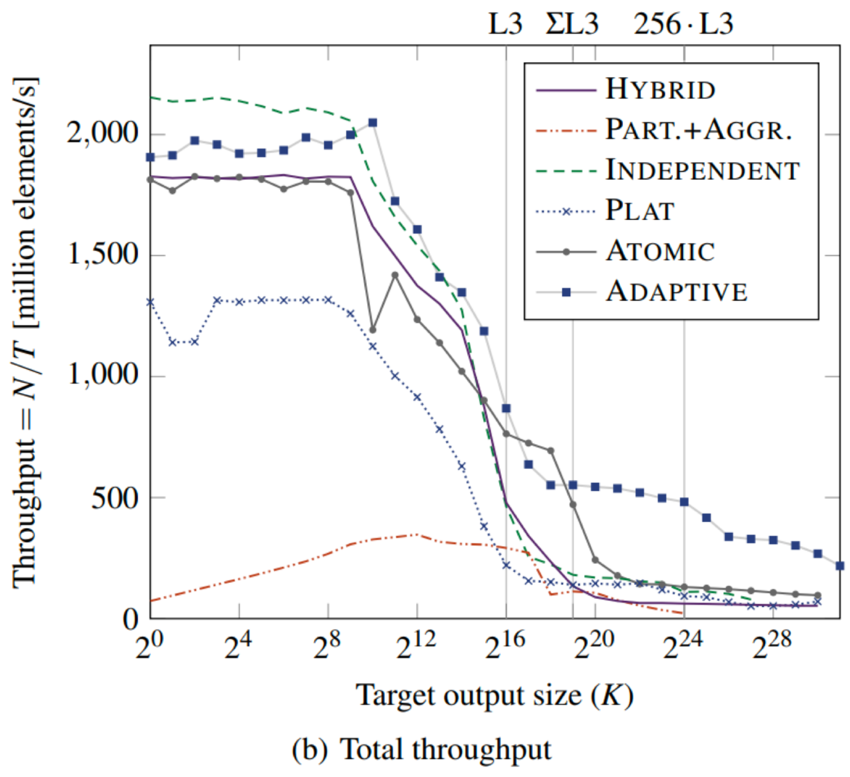
Figure 7: Scalability of ADAPTIVE with the number of columns using $P = 20$ threads.

Figure 7 shows how the number of aggregate columns affects the performance of ADAPTIVE for different output cardinalities K. Indeed the plot indicates that the run time per element is almost the same for any number of columns.

Benchmark with previous work



(a) Single core run time per element



(b) Total throughput

Figure 8: Comparison with prior work of Cieslewicz and Ross [13] and Ye et al. [46] using $P = 20$ threads.

ADAPTIVE achieves a speedup of at least factor 2.7 for all K ≤ 221

ADAPTIVE is also as least as fast as almost all other algorithm for other values of K

Skew resistance

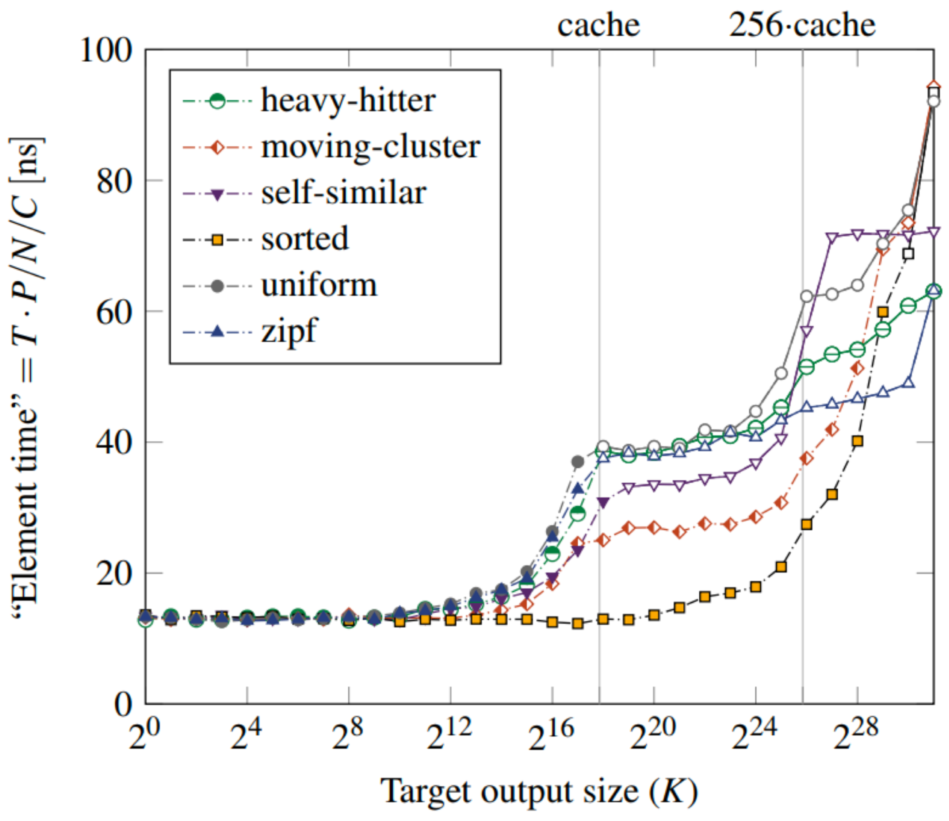


Figure 9: ADAPTIVE on different data sets using $P = 20$ threads.

We now extend the experiments on uniform data to other data sets in order to test the skew resistance of our ADAPTIVE operator.

Figure 9 shows the performance of ADAPTIVE on all data sets. The first and most important observation is that ADAPTIVE is not slower on the other distribution than uniform. In this sense, uniform is the hardest distribution for our operator and skew only improves its performance. Since skew means that some keys occur more often than others, our operator can benefit from skew by using hashing for early aggregation of these values.

Conclusion

In summary, our work starts with the assumption that even in the in memory setting, the movement of data is the hard part of relational operators such as aggregation.

- We use an external memory model to show that HASHAGGREGATION and SORTAGGREGATION are equivalent in terms of the number of cache lines transfers they incur.
- Consequently we design an algorithmic framework based on sorting by hash value that allows to combine hashing for early aggregation and integer sorting routines depending on the locality of the data. We tune both the hashing and the sorting to switch between the two.
- We show extensive experiments on different data sets and a comparison with several algorithms from prior work. We are able to outperform all our competitors by up to factor 3.7.