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.

- 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: (0100, b, 3) (0010, a, 7) (1110, c, 2) (0100, b, 4) (1100, e, 3) (0100, b, 6) |
|---------------------------------|---------------------------------|---------------------------------|
| (hash, group, value)            | (hash, group, value)            | (hash, group, value)            |

**1st level of recursion**

<table>
<thead>
<tr>
<th>Hashing</th>
<th>Partitioning</th>
</tr>
</thead>
<tbody>
<tr>
<td>hash 1:</td>
<td>partitions:</td>
</tr>
<tr>
<td>0010, a, 7</td>
<td>(0100, b, 2) (0100, b, 5)</td>
</tr>
<tr>
<td>0100, b, 7</td>
<td>(1110, c, 2)</td>
</tr>
<tr>
<td>0100, b, 6</td>
<td></td>
</tr>
<tr>
<td>(hash, group, value)</td>
<td>(hash, group, value)</td>
</tr>
</tbody>
</table>

**2nd level of recursion**

<table>
<thead>
<tr>
<th>hash range “0j”</th>
<th>hash range “1j”</th>
</tr>
</thead>
<tbody>
<tr>
<td>hash table (part):</td>
<td>hash table (part):</td>
</tr>
<tr>
<td>0010, a, 7</td>
<td>0100, b, 20</td>
</tr>
<tr>
<td>0100, b, 20</td>
<td>1100, e, 3</td>
</tr>
<tr>
<td>1100, e, 3</td>
<td></td>
</tr>
</tbody>
</table>

- 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)ranged 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”.

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 hashing as early as possible.

4. Evaluation: Comparison with prior work

**Result**:

- Our algorithm (“Adaptive”) faster than all others \[1,2\] for \(K > 2^{10}\).
- Up to factor 3.7 speedup to second best.


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?