Engineering Aggregation Operators for Relational In-Memory Database Systems

Ingo Müller – PhD defense – February 11, 2016
Introduction – The Race of Database Systems

- Trend 1: data volumes increase exponentially (or faster)
- Trend 2: compute power increases exponentially
  - But also more and more complex, for example memory access

Database systems are in a continuous race to translate Moore’s law.
### Introduction – Grouping with Aggregation

#### Input (Sales)

<table>
<thead>
<tr>
<th>Store</th>
<th>Item</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Berlin</td>
<td>pen</td>
<td>1.00€</td>
</tr>
<tr>
<td>Berlin</td>
<td>paper</td>
<td>3.00€</td>
</tr>
<tr>
<td>Paris</td>
<td>ruler</td>
<td>2.00€</td>
</tr>
<tr>
<td>Berlin</td>
<td>pen</td>
<td>1.00€</td>
</tr>
<tr>
<td>Paris</td>
<td>pen</td>
<td>1.00€</td>
</tr>
<tr>
<td>Vienna</td>
<td>paper</td>
<td>3.00€</td>
</tr>
</tbody>
</table>

#### Output

\[
\sum \text{SUM(Price)}
\]

<table>
<thead>
<tr>
<th>Store</th>
<th>Item</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Paris</td>
<td>3.00€</td>
<td></td>
</tr>
<tr>
<td>Vienna</td>
<td>3.00€</td>
<td></td>
</tr>
<tr>
<td>Berlin</td>
<td>5.00€</td>
<td></td>
</tr>
</tbody>
</table>

What is the sum of the prices of all sold items per store?

\[
\text{SELECT Store, SUM(Price) AS Sum FROM Sales GROUP BY Store}
\]
Challenges and Overview

- **Cache efficiency**
  - lower bound + (optimal) recursive algorithm*
- **Optimizer independence**
  - adaptive execution strategy*
- **Memory constraint**
  - intra-operator pipelining
- **CPU friendliness**
  - low-level tuning of inner loops*
- **Parallelism**
  - work stealing*
- **Skewed data distribution**
  - robust algorithm design*
- **Communication efficiency**
  - adaptive pre-aggregation
- **System integration**
  - compatible with major DB architectures*

* [SIGMOD15]

Result: up to 3.7x faster and robust enough for use in production.
Challenge: Cache Efficiency – Motivation

Two textbook algorithms:

- **Hash-Aggregation**
  - Insert every row into hash map with grouping attributes as key
  - Aggregate to existing intermediate result

- **Sort-Aggregation**
  - Sort input by grouping attributes
  - Aggregate consecutive rows in a single pass

Can we do better? Long standing conjecture: no!
External Memory Model – Proof Techniques

- Known lower bounds for Aggregation
  - Based on comparisons [MR91, AK+93]
    → Do not hold for Hashing!

- Proof technique [AV88, Gre12]
  - Count the number of possible permutations after $t$ transfers
  - Compare with possible number of input permutations

- Modifications for Aggregation
  - Allow semi-group operation in cache
  - Count “permutations” as before
External Memory Model – Result

- Lower bound* for Aggregation
  \[ \frac{N}{PB} \left[ \log_M \frac{K}{B} \right] \text{ block transfers} \]
  
  *simplified asymptotic worst case

- Same bound as for Sorting Multisets [AK+93]

\[ N \quad M = \text{cache size} \]
\[ B \quad B = \text{block size} \]
\[ K \quad N = \text{input size} \]
\[ P \quad K = \text{output size} \]

We confirm: Aggregation is as hard as Sorting! → Use as guideline.
Outline

- **Cache efficiency**
  - lower bound
  - (optimal) recursive algorithm

- **Optimizer independence**
  - adaptive execution strategy

- **Memory constraint**
  - intra-operator pipelining
Challenge: Adaptivity – Motivation

- Traditional approach [Gra93]
  - Implement HashAggregation and SortAggregation
  - Optimizer selects implementation based on statistics beforehand

- Problem
  - Wrong statistics may lead to suboptimal performance

Our goal: adaptively switch between Hashing and Sorting during execution.
Adaptivity – Mixing Hashing and Sorting

Recursive algorithm:

- In each level of recursion: mix **Hashing** and **Sorting** adaptively
- **Partitioning** recurses when necessary
- **Hashing** ends recursion when possible efficiently

![Diagram showing the recursive algorithm](image)
Our mechanism achieves the best of Hashing and Sorting.
Efficient recursive processing is crucial for large outputs.

2 Xeon E7-8870 CPUs (each 10 cores)
N = $2^{32}$, uniform distribution

Original implementation of [CR07,YR+11]
Outline

- Cache efficiency
  - lower bound
  - (optimal) recursive algorithm

- Optimizer independence
  - adaptive execution strategy

- Memory constraint
  - intra-operator pipelining
Memory Constraint – Intra-Operator Pipelining

- Split work into blocks
- Limit number of blocks
- Recycle free blocks
- Interleave/Overlap processing levels

Pipelining allows to limit the amount of intermediate memory.
Memory Constraint – Intra-Operator Scheduling

- In which level to work?
  → Heuristic: target 50% memory usage

- On which partition to work?
  → Priority queue on partition length
Memory Constraint – Evaluation

Performance basically preserved (for moderate result sizes)

Trade-off between memory usage and performance

Cache efficiency can be achieved under memory constraint.

Input size = 16GiB, memory constraint = 256MiB

Input size = 16GiB, K = 2^{23}
Summary

- **Cache efficiency**
  - lower bound + (optimal) recursive algorithm

- **Optimizer independence**
  - adaptive execution strategy

- **Memory constraint**
  - intra-operator pipelining

- **CPU friendliness**
  - low-level tuning of inner loops

- **Parallelism**
  - work stealing

- **Skewed data distribution**
  - robust algorithm design

- **Communication efficiency**
  - adaptive pre-aggregation

- **System integration**
  - compatible with major DB architectures

Thank you! Questions?

* [SIGMOD15]
References


The photo of the title page has been provided by Vladislav Bezrukov under the [CC BY 2.0 licence](http://creativecommons.org/licenses/by/2.0/). It has not been modified.
Backup – List of Publications


Backup – Columnwise Processing Scheme

- Grouping column
- Mapping vector
- Aggregate column

Ingo Müller – PhD Defense
Engineering Aggregation Operators for Relational In-Memory Database Systems
Backup – Scaling with the Number of Columns

"Element time" = T · P/N/C [ns]

Number of columns (C)

- K = 2^{27}
- K = 2^{23}
- K = 2^{19}
- K = 2^{15}
- K = 2^{11}
Backup – Scaling with the Number of Cores

![Graph showing the speedup with the number of cores](image-url)

- **optimal**
- $K = 2^{31}$
- $K = 2^{26}$
- $K = 2^{21}$
- $K = 2^{16}$
- $K = 2^{11}$

**Y-axis**: Speedup

**X-axis**: Number of cores ($P$)
Backup – Workload Study

Number of output rows (K)

2^25
2^20
2^15
2^10
2^5

Number of input rows (N)

2^0
2^4
2^8
2^12
2^16
2^20
2^24
2^28

- customer (small)
- customer (big)
- TPC-H SF100
- input = output
Backup – Skewed Data Distributions

"Element time" = T \cdot P/N/C [ns]

- heavy-hitter
- moving-cluster
- self-similar
- sorted
- uniform
- zipf

Target output size (K)

cache 256-cache