Applicability of GPU Computing for Efficient Merge in In-Memory Databases

Jens Krueger, Martin Grund, Ingo Jaeckel, Alexander Zeier, Hasso Plattner
Hasso Plattner Institute for IT Systems Engineering
University of Potsdam
Potsdam, Germany
Introduction (1)

- Enterprise applications have evolved: not just OLAP vs. OLTP
- Range selects occur often
- Real world is more complicated than single tuple access
Introduction (2)

- Enterprise data is wide and sparse
- Most columns are empty or have a low cardinality of distinct values
- Sparse distribution facilitates high compression
System Overview

• Based on HYRISE: In-memory compressed vertical partitionable database engine
  • Completely in main memory
  • Organizes data column-oriented
  • Applies dictionary compression with a order-preserving dictionary and a bit-compressed attribute vector
  • Uses a differential store concept to support data modifications

• Efficiently executes both OLTP and OLAP requests on structured enterprise data
**Terminology**

- **Table**: A relation table with NC columns, with one write-optimized (delta) and one read-optimized (main) partition.

- **Update**: Any modification operation on the table resulting in an entry in the delta partition.

- **Main Partition**: Compressed and read-optimized part of the column. Consists of an order-preserving dictionary and an attribute vector with bit-compressed value ids.

- **Delta Partition**: Uncompressed write-optimized part of the column where all updates are stored until the merge process is completed.

- **Merge Process**: Applies compression to delta and main partition to create new main partition.
System Overview (2)
Merge Process

• Transfer **updates** from uncompressed delta partition into main partition
• Requirements
  o has to be performed while the system is operational, hence works on a copy
  o minimal time of increased resource utilization
• Phases:
  • Prepare merge
  • Attribute merge
    1. Merge dictionaries
       1.a Build delta dictionary
       1.b Merge main and delta dictionary
    2. Update compressed values
       2.a Compute new compressed value length
       2.b Create new compressed main
  • Commit merge
Attribute Merge

For the $j$-th column, the input for the merging algorithm consists of $M^j$, $D^j$ and $U_M^j$, while the output consists of $M'^j$ and $U_M'^j$.

Runtime complexity:

Step 1: $|U_M'| = |D^j \cup U_M^j|$

Step 2: $N_M = N_M + N_D$

As Step 2 is already bandwidth bound [1], we focus on Step
Motivation / Trade-offs

- GPUs offer up to two orders of magnitude more cores than a CPU
  - Increases the maximum possible speedup through parallelization accordingly

- **But:** the in-/output data needs to be transferred over the PCI-Express bus which has a limited bandwidth
  - to be faster, GPU implementations have to be finished before CPU implementations **including the data transfers**
NVIDIA Thrust

- *Thrust* is a CUDA library of parallel algorithms resembling the C++ STL
- **Assumption:** An implementation that uses operations provided by a mature CUDA library can provide better performance than a custom-made CUDA kernel
  - e.g. thrust::sort, thrust::unique, thrust::reduce, thrust::lower_bound

<table>
<thead>
<tr>
<th>Year</th>
<th>Device</th>
<th>Rate</th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>2009</td>
<td>GTX 280</td>
<td></td>
<td></td>
<td>200</td>
</tr>
<tr>
<td>2010</td>
<td>Knights Ferry vs. GTX 280</td>
<td>560</td>
<td>176</td>
<td></td>
</tr>
<tr>
<td>2010</td>
<td>Core i7 vs. GTX 280</td>
<td>250</td>
<td>200</td>
<td></td>
</tr>
<tr>
<td>2009</td>
<td>Tesla C1060</td>
<td></td>
<td>300</td>
<td></td>
</tr>
<tr>
<td>2010</td>
<td>GTX 285</td>
<td></td>
<td>550</td>
<td></td>
</tr>
<tr>
<td>2011</td>
<td>GTX 480</td>
<td></td>
<td>1005</td>
<td></td>
</tr>
</tbody>
</table>
GPU Duplicate Removal

- **Trade-off:** Delta partition insert costs vs. costs for creating a delta dictionary during merge process
- Assuming that inserting into the CSB+ structure is too expensive in insert/update-intensive workloads
- Without the CSB+ structure duplicates have to be removed to create a dictionary
Duplicate Removal (2)

- Remove duplicates by sorting and removing subsequent duplicates with `thrust::sort` and `thrust::unique`
- Up to 27 times faster than naïve `std::sort` and `std::unique`
Dictionary Merge

• Propose a custom kernel for merging dictionaries
  o Block-Wise Parallel Slice Merge (BWS)
• ... And *Thrust*-supported approaches that reuses the duplicate removal approach
  o Concatenate-Sort-Unique-Binary Search (CSUBS)
  o Merge-Unique-Binary-Search (MUBS)
BWS Merge

- Merge two sorted lists
- All values in a list are distinct
- **But:** a value can appear in both lists at the same time

**Idea:**
- Partition both input lists into slices
- All values of a slice are smaller than the values of the subsequent slice
- Static partitioning is not sufficient since it allows duplicates
BWS Merge (2)

• First list: partition into equally sized slices
• Second list: determine boundaries of the slices with binary search

• Partition both dictionaries (CPU)
• Merge slices (GPU)
  o Determine number of unique values per thread block
  o Inform other threads of local unique value count
  o Write unique values

• Concatenate block-wise output (CPU)
• Use parallel binary search to fill auxiliary structures or: set them on the GPU
CSUBS Merge

- **Concatenate, Sort, Unique, Binary Search**

- Use *Thrust primitives* to implement dictionary merge:
  - Concatenate dictionaries in GPU memory with `thrust::copy`
  - Sort concatenated dictionaries with `thrust::sort`
  - Remove subsequent duplicates with `thrust::unique`
  - Map values to their new position with `thrust::lower_bound`
    - create auxiliary structures with binary search for each value of both dictionaries
MUBS Merge

- **Merge, Unique, Binary Search**
- CSUBS approach does not exhibit the fact that both lists are already sorted
- Rather than concatenating and sorting use *Thrust*'s merge primitive
  - Merge both sorted lists into a new list `thrust::merge`
  - Remove subsequent duplicates with `thrust::unique`
  - Map values to their new position with `thrust::lower_bound`
Evaluation - Environment

- GPU: Tesla C2050 GPU, 3GB memory
- CPU: single core of a Xeon E5620 processor as baseline
  - Used STL implementations, e.g. `std::sort`, `std::unique`, and the default merge implementation applied in HYRISE
- Data:
  - Single column
  - 32-bit integer values and
  - Strings with a length of up to eight characters
Evaluation - Results

• Numbers:
  o Dictionary merge approaches are up to 40% faster
  o Duplicate removal is up to 27 times faster
  o Page-locked memory increases throughput by up to 10%

• Strings:
  o Throughput of all implementations is reduced
  o BWS and MUBS outperform the CPU implementation
    (sorting strings on a GPU is expensive)
Evaluation - Breakdown

- Relative run-time of individual dictionary merge steps
Conclusion

- Architecture conscious optimizations are needed
- Merge run-time can be reduced with a GPU implementation
  - 27 times improvement on duplicate removal
  - 40% speed up on dictionary merge
- Data transfer is still a bottleneck
- String processing is expensive
- Limited global memory of the GPU compared to main memory
Thank You!

sources:
Backup
Future Work

- Performance analysis for different data
  - Which speedup can be achieved for which data characteristics?

- Support for large numbers of distinct values

- More elaborate scheduling scheme
  - e.g. dynamic scheduling

- Dedicated merge server(s)
  - receives *merge tasks*, responds *merged tables*
  - may be shared across multiple databases
  - e.g. server with few CPU cores but many GPUs
merge process