Exploring Query Execution Strategies for JIT, Vectorization and SIMD

Tim Gubner, Peter Boncz

CWI

September 1, 2017
Motivation

- What will future DB engines look like?
- New hardware increasingly heterogenous: GPUs, FPGAs on the same chip, specialized CPU instructions
- For performance: exploit hardware features
- → Multi-dimensional design space
Contribution

- Shed some light into the design space
- Compare multiple implementations of TPC-H Q1:
  - Tuple-at-a-time / block-at-a-time
  - Regular / compact data types
  - Overflow checking / prevention
  - Different aggregation techniques
  - Varying aggregation table layout
**SIMD (Single Instruction Multiple Data)**

- One instruction processes 512 bits (AVX-512)
- Less bits per element = more elements processed per instruction

<table>
<thead>
<tr>
<th>Data type width</th>
<th>Parallelism</th>
</tr>
</thead>
<tbody>
<tr>
<td>64-bit</td>
<td>8</td>
</tr>
<tr>
<td>32-bit</td>
<td>16</td>
</tr>
<tr>
<td>16-bit</td>
<td>32</td>
</tr>
<tr>
<td>8-bit</td>
<td>64</td>
</tr>
<tr>
<td>1-bit</td>
<td>512</td>
</tr>
</tbody>
</table>
Data types

- Derived from schema:
  DECIMAL(15,2) → 64-bit integer (32 < $\log_2(10^{15}) < 64$)
- Thinner data types → more data items per clock cycle
Compact data types by example

- Reduce size of data types based on actual data
- Example from TPC-H:
  - Schema: \( tax \in \text{DECIMAL}(15, 2) \rightarrow 64\text{-bit integer} \)
  - Data: \( tax \in \{0.0, 0.1, \ldots, 0.8\} \rightarrow [0, 80] \rightarrow 8\text{-bit integer} \)
  - Computation: \( tax \cdot tax \rightarrow 8\text{-bit integer} \cdot 8\text{-bit integer} = 16\text{-bit integer} \)
- Restrict data types using domain minimum & maximum
Overflow handling

- Overflow handling is required to guarantee correctness

<table>
<thead>
<tr>
<th>Detection</th>
<th>Prevention</th>
</tr>
</thead>
<tbody>
<tr>
<td>Check</td>
<td>Larger data types, overflow cannot (realistically) happen</td>
</tr>
<tr>
<td>via overflow CPU flag</td>
<td></td>
</tr>
<tr>
<td>or specific code</td>
<td></td>
</tr>
</tbody>
</table>

- Overflow checking code is inefficient
- Overflow CPU flag is not SIMD friendly
- → Overflow prevention
Case study: TPC-H Q1

```sql
SELECT
  l_returnflag, l_linestatus,
  count(*) AS count_order,
  sum(l_quantity) AS sum_qty,
  avg(l_quantity) AS avg_qty,
  avg(l_discount) AS avg_disc,
  avg(l_extendedprice) AS avg_price,
  sum(l_extendedprice) AS sum_base_price,
  sum(l_extendedprice*(1-l_discount)) AS sum_disc_price,
  sum(l_extendedprice*(1-l_discount)*(1+l_tax)) AS sum_charge
FROM
  lineitem
WHERE
  l_shipdate <= date '1998-12-01' - interval '90' day
GROUP BY
  l_returnflag, l_linestatus
ORDER BY
  l_returnflag, l_linestatus
```
<table>
<thead>
<tr>
<th>Expression</th>
<th>Regular</th>
<th>Compact</th>
</tr>
</thead>
<tbody>
<tr>
<td>l_tax</td>
<td>64</td>
<td>8</td>
</tr>
<tr>
<td>l_returnflag</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>l_linestatus</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>l_extendedprice</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>1+l_tax</td>
<td>64</td>
<td>8</td>
</tr>
<tr>
<td>(1-l_discount)*(1+l_tax)</td>
<td>64</td>
<td>16</td>
</tr>
<tr>
<td>l_extendedprice*(1-l_discount)</td>
<td>64</td>
<td>32</td>
</tr>
<tr>
<td>l_extendedprice*(1-l_discount)*(1+l_tax)</td>
<td>64</td>
<td>64</td>
</tr>
</tbody>
</table>
## Q1 flavors

<table>
<thead>
<tr>
<th>Base flavor</th>
<th>Intermediate storage</th>
<th>Selection strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>X100</td>
<td>CPU cache</td>
<td>Sel. vector</td>
</tr>
<tr>
<td>HyPer</td>
<td>Registers</td>
<td>Branching</td>
</tr>
<tr>
<td>Handw. AVX-512</td>
<td>Registers</td>
<td>Bit mask</td>
</tr>
</tbody>
</table>
Design space explored

- **X100**
  - Data types: Full, Compact
  - Aggregate table layout: NSM (row-wise), DSM (column-wise)
  - Aggregation algorithm: Standard, Standard Fused, In-Reg

- **HyPer**
  - Data types: Full, Compact
  - Overflow: Default (set flag), OverflowBranch (branching), NoOverflow (prevention)
Q1 flavors on Sandy Bridge

- X100 Full NSM Standard
- X100 Full DSM Standard
- X100 Full NSM Standard Fused
- X100 Full NSM In-Reg
- X100 Compact NSM Standard
- X100 Compact DSM Standard
- X100 Compact NSM Standard Fused
- X100 Compact NSM In-Reg
- HyPer Full
- HyPer Full OverflowBranch
- HyPer Full NoOverflow
- HyPer Compact
- HyPer Compact OverflowBranch
- HyPer Compact NoOverflow
- Weld

Runtime in seconds
Q1 flavors on Knights Landing

- X100 Full NSM Standard
- X100 Full DSM Standard
- X100 Full NSM Standard Fused
- X100 Full NSM In-Reg
- X100 Compact NSM Standard
- X100 Compact DSM Standard
- X100 Compact NSM Standard Fused
- X100 Compact NSM In-Reg
- X100 Compact NSM In-Reg AVX-512
- HyPer Full
- HyPer Full OverflowBranch
- HyPer Full NoOverflow
- HyPer Compact
- HyPer Compact OverflowBranch
- HyPer Compact NoOverflow
- Weld
- Handwritten AVX-512
- Handwritten AVX-512 Only64BitAggr

Runtime in seconds
Takeaways

- Do not rely on schema!
- Exploit data statistics instead
- Ingredients for performance:
  - Compact data types
  - Overflow prevention
  - Adapt algorithms to take advantage of compact data types
Future work

- Even thinner data types / different representations → compressed execution
- System that generates such a query
Questions?