Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation †

Harald Lang¹, Tobias Mühlbauer¹, Florian Funke²,*, Peter Boncz³,*, Thomas Neumann¹, Alfons Kemper¹

¹Technical University Munich, ²Snowflake Computing, ³Centrum Wiskunde & Informatica

†To appear at SIGMOD 2016. * Work done while at Technical University Munich.
Goals

▶ Primary goal
  ▶ Reducing the memory-footprint in hybrid OLTP&OLAP database systems
  ▶ Retaining high query performance and transactional throughput

▶ Secondary goals / future work
  ▶ Eviting cold data to secondary storage
  ▶ Reducing costly disk I/O

▶ Out of scope
  ▶ Hot/cold clustering (see previous work of Funke et al.: “Compacting Transactional Data in Hybrid OLTP&OLAP Databases”)
Compression in Hybrid OLTP&OLAP Database Systems

- SAP HANA (existing approach)
  - Compress entire relations
  - Updates are performed in an uncompressed write-optimized partition
  - Implicit hot/cold clustering
  - Merge partitions

- HyPer (our approach)
  - Split relations in fixed size chunks (e.g., 64 K tuples)
  - Cold chunks are “frozen” into immutable Data Blocks
Data Blocks

- Compressed columnar storage format
  - Designed for cold data (mostly read)
  - Immutable and self-contained
  - Fast scans and fast point-accesses
  - Novel index-structure to narrow scan ranges
Compression Schemes

- Lightweight compression only
  - Single value, byte-aligned truncation, ordered dictionary
- Efficient *predicate evaluation, decompression* and *point-accesses*
- Optimal compression chosen based on the actual value distribution
  - Improves compression ratio, amortizes light-weight compression schemes and redundancies caused by block-wise compression
Positional SMAs

- Lightweight indexing
- Extension of traditional SMAs (min/max-indexes)
- Narrow scan ranges in a Data Block

Supported predicates:

- `column \circ constant`, where `\circ \in \{=, is, <, \le, \ge, >\}`
- `column between a and b`
Positional SMAs - Details

- Lookup table where each table entry contains a range with potential matches
- For $n$ byte values, the table consists of $n \times 256$ entries
- Only the *most significant non-zero byte* is considered
Positional SMAs - Details

- Lookup table where each table entry contains a range with potential matches
- For $n$ byte values, the table consists of $n \times 256$ entries
- Only the *most significant non-zero byte* is considered

![Diagram of positional SMAs]

- **lookup table**
  - 256 range entries
  - 256 range entries
  - [0,3]
  - 256 range entries

- **pos:**
  - 0
  - 1
  - 2
  - 3

- **data:**
  - 0x0200AA
  - 0x0000AE
  - 0x02FA42
  - ...

- **preferred range:** achieved by using the delta value – SMAMin

- **max # of values sharing an entry:**
  - 1
  - $2^8$
  - $2^{16}$

- **range:** [0,3)

- **most significant non-zero byte**

- **leading zero-bytes**

- **tail bytes**
Positional SMAs - Example

SMA min: 2
SMA max: 999

probe 7

delta = 5 (7-min)

bytes of delta

00 00 00 05

5 (leading non-0 byte)

delta = 5 (7-min)

0x03 + 256 = 259
(second byte)

probe 998

delta = 996 (998-min)

00 00 03 E4

bytes of delta

lookup table

[0,6)
[1,17)
[16,19)
[6,7)
[0,0)
Challenge for JIT-compiling Query Engines

- HyPer compiles queries just-in-time (JIT) using the LLVM compiler framework
- Generated code is *data-centric* and processes a *tuple-at-a-time*

```cpp
for (const Chunk& c : relation.chunks) {
    for (unsigned row=0; row!=c.rows; ++row) {
        auto attr0 = c.column[0].data[row];
        auto attr3 = c.column[3].data[row];
        // check scan restrictions
        if (tuple qualifies) {
            // code of consuming operator
            ...
        }
    }
}
```

- Data Blocks individually determine the best suitable compression scheme for each column on a per-block basis
- The *variety of physical representations* either results in
  - multiple code paths => exploding compile-time
  - or interpretation overhead => performance drop at runtime
Vectorization to the Rescue

- Vectorization greatly reduces the interpretation overhead
- Specialized vectorized scan functions for each compression scheme
- Vectorized scan extracts matching tuples to temporary storage where tuples are consumed by tuple-at-a-time JIT code
Predicate Evaluation using SIMD Instructions

Find Initial Matches

- aligned data
- unaligned data
- read offset
- remaining data
- predicate evaluation
- movemask
  - precomputed positions table
  - lookup
  - add global scan position and update match vector
  - write offset
  - match positions
Predicate Evaluation using SIMD Instructions

Additional Restrictions

match positions: 1, 3, 14, -,-,-,-,-
read offset
movemask
precomputed positions table: 255 ... 172 ... 1... 0
0, 2, ... 20, 25, 26, -,-,-,-
shuffle match vector
store
match positions: 17, 18, 20, 21, 25, 26, 29, 31

write offset
gather
predicate evaluation
lookup
movemask = \(172_{10}\)
shuffle match vector
store
Evaluation
## Compression Ratio

Size of *TPC-H, IMDB cast info*, and a *flight* database in HyPer and Vectorwise:

<table>
<thead>
<tr>
<th></th>
<th>TPC-H SF100</th>
<th>IMDB(^1) cast info</th>
<th>Flights(^2)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>uncompressed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CSV</td>
<td>107 GB</td>
<td>1.4 GB</td>
<td>12 GB</td>
</tr>
<tr>
<td>HyPer</td>
<td>126 GB</td>
<td>1.8 GB</td>
<td>21 GB</td>
</tr>
<tr>
<td>Vectorwise</td>
<td>105 GB</td>
<td>0.72 GB</td>
<td>11 GB</td>
</tr>
<tr>
<td></td>
<td>compressed</td>
<td></td>
<td></td>
</tr>
<tr>
<td>HyPer</td>
<td>66 GB</td>
<td>(0.62×)</td>
<td>0.50 GB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(0.36×)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>4.2 GB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(0.35×)</td>
</tr>
<tr>
<td>Vectorwise</td>
<td>54 GB</td>
<td>(0.50×)</td>
<td>0.24 GB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(0.17×)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>3.2 GB</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(0.27×)</td>
</tr>
</tbody>
</table>

\(^1\) [http://www.imdb.com](http://www.imdb.com)  
## Query Performance

Runtimes of TPC-H queries (scale factor 100) using different scan types on uncompressed and compressed databases in HyPer and Vectorwise.

<table>
<thead>
<tr>
<th>scan type</th>
<th>geometric mean</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>HyPer</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>JIT (uncompressed)</td>
<td>0.586s</td>
<td>21.7s</td>
</tr>
<tr>
<td>Vectorized (uncompressed)</td>
<td>0.583s (1.01×)</td>
<td>21.6s</td>
</tr>
<tr>
<td>+ SARG</td>
<td>0.577s (1.02×)</td>
<td>21.8s</td>
</tr>
<tr>
<td>Data Blocks (compressed)</td>
<td>0.555s (1.06×)</td>
<td>21.5s</td>
</tr>
<tr>
<td>+ SARG/SMA</td>
<td>0.466s (1.26×)</td>
<td>20.3s</td>
</tr>
<tr>
<td>+ PSMA</td>
<td>0.463s (1.27×)</td>
<td>20.2s</td>
</tr>
<tr>
<td><strong>Vectorwise</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>uncompressed storage</td>
<td>2.336s</td>
<td>74.4s</td>
</tr>
<tr>
<td>compressed storage</td>
<td>2.527s (0.92×)</td>
<td>78.5s</td>
</tr>
</tbody>
</table>
Speedup of TPC-H Q6 (scale factor 100) on block-wise sorted\(^3\) data (+SORT).

\(^3\)sorted by l_shipdate
Throughput (in lookups per second) of random point access queries

\[
\text{select } * \text{ from customer where c_custkey} = \text{randomCustKey()}
\]

on TPC-H scale factor 100 with a primary key index on \text{c_custkey}.

<table>
<thead>
<tr>
<th>Throughput [lookups/sec]</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Uncompressed</td>
<td>545,554</td>
</tr>
<tr>
<td>Data Blocks</td>
<td>294,291 (0.54 \times)</td>
</tr>
</tbody>
</table>
OLTP Performance - TPC-C

TPC-C transaction throughput (5 warehouses), old neworder records compressed into Data Blocks:

<table>
<thead>
<tr>
<th>Throughput [Tx/sec]</th>
<th>Uncompressed</th>
<th>Data Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>89,229</td>
<td>88,699</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0.99 ×)</td>
</tr>
</tbody>
</table>

Only read-only TPC-C transactions order status and stock level; all relations frozen into Data Blocks:

<table>
<thead>
<tr>
<th>Throughput [Tx/sec]</th>
<th>Uncompressed</th>
<th>Data Blocks</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>119,889</td>
<td>109,649</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(0.91 ×)</td>
</tr>
</tbody>
</table>
Performance of SIMD Predicate Evaluation

Speedup of SIMD predicate evaluation of type \( l \leq A \leq r \) with selectivity 20%:
Performance of SIMD Predicate Evaluation (cont’d)

Costs of applying an additional restriction with varying selectivities of the first predicate and the selectivity of the second predicate set to 40%:

![Graphs showing the costs of applying an additional restriction with varying selectivities of the first predicate and the selectivity of the second predicate set to 40% for 8-bit, 16-bit, 32-bit, and 64-bit operations.](image)
Advantages of Byte-Addressability

Predicate Evaluation

Cost of evaluating a SARGable predicate of type $l \leq A \leq r$ with varying selectivities:

$\text{dom}(A) = [0, 2^{16}]$

Intentionally, the domain exceeds the 2-byte truncation by one bit

17-bit codes with bit-packing, 32-bit codes with Data Blocks
Advantages of Byte-Addressability

Unpacking matching tuples

Cost of unpacking matching tuples:

- 3 attributes, \( \text{dom}(A) = \text{dom}(B) = [0, 2^{16}] \) and \( \text{dom}(C) = [0, 2^8] \)
- Intentionally, the domains exceed 1-byte and 2-byte truncation by one bit
- The compression ratio of bit-packing is almost two times higher in this scenario
Thank you!