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

Harald Lang[1], Tobias Mühlbauer[1], Florian Funke[2,*],
Peter Boncz[3,*], Thomas Neumann[1], Alfons Kemper[1]

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

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

Goals

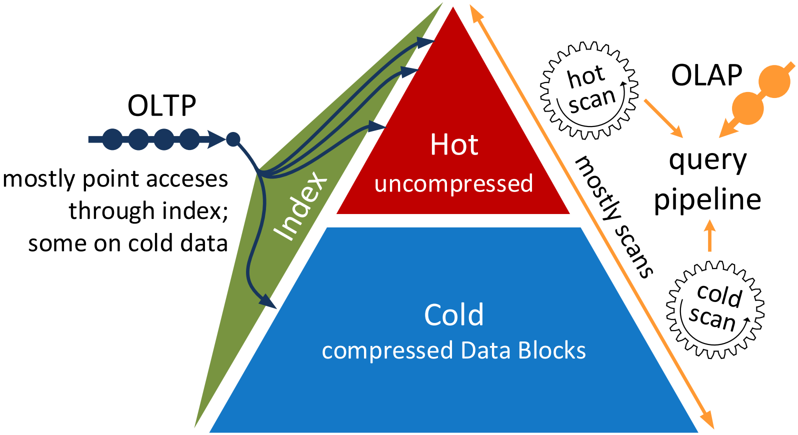
- Primary goal
  - Reducing the memory-footprint in hybrid OLTP&OLAP database systems
  - Retaining high query performance and transactional throughput
- Secondary goals / future work
  - Eviting cold data to secondary storage
  - Reducing costly disk I/O
- Out of scope
  - Hot/cold clustering (see previous 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

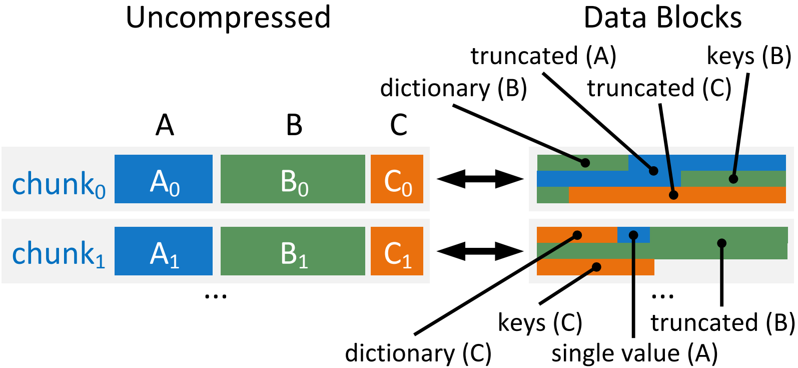
Data Blocks

- 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



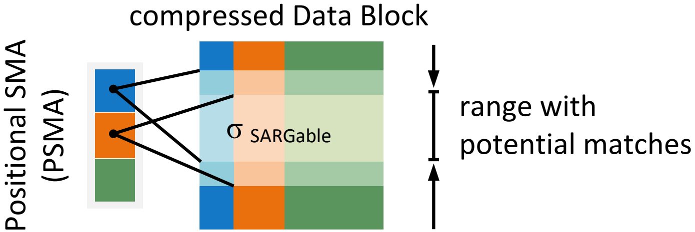
Compression Schemes

- 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

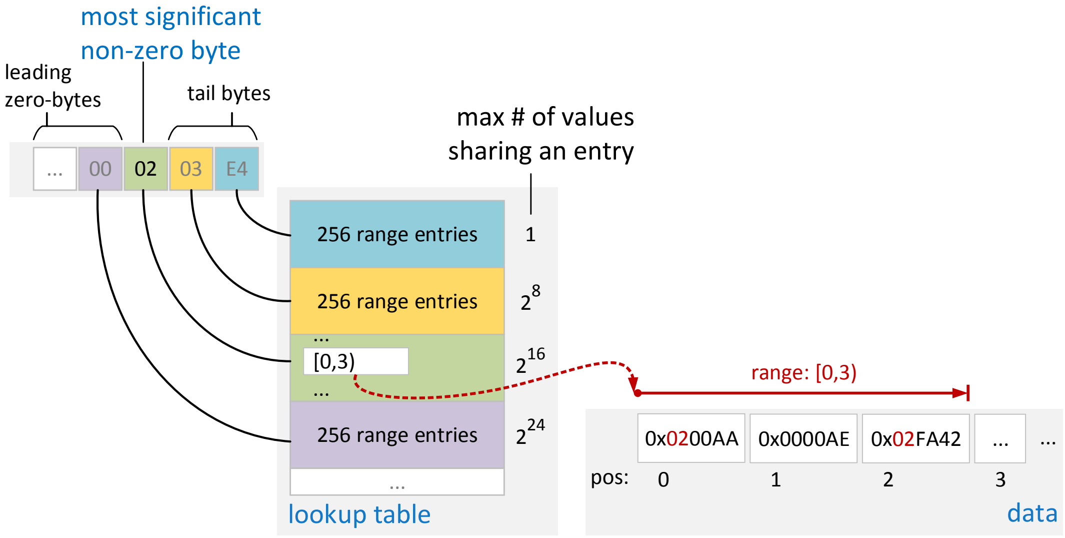
- Lightweight indexing
- Extension of traditional SMAs (min/max-indexes)
- Narrow scan ranges in a Data Block



- Supported predicates:
  - $column \circ constant$, where $\circ \in \{=, is, <, \leq, \geq, >\}$
  - $column$ **between** a **and** b

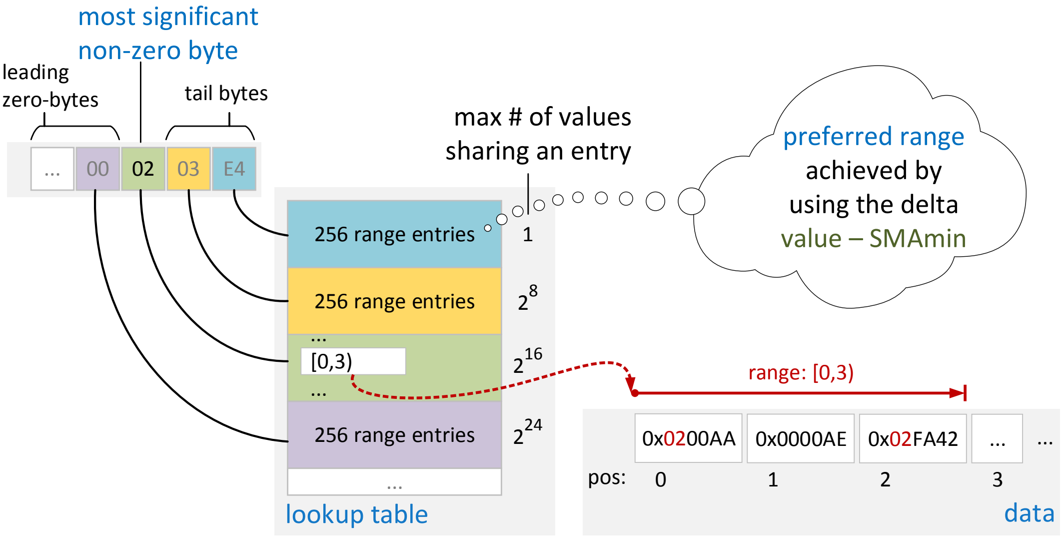
Positional SMAs - Details

- 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

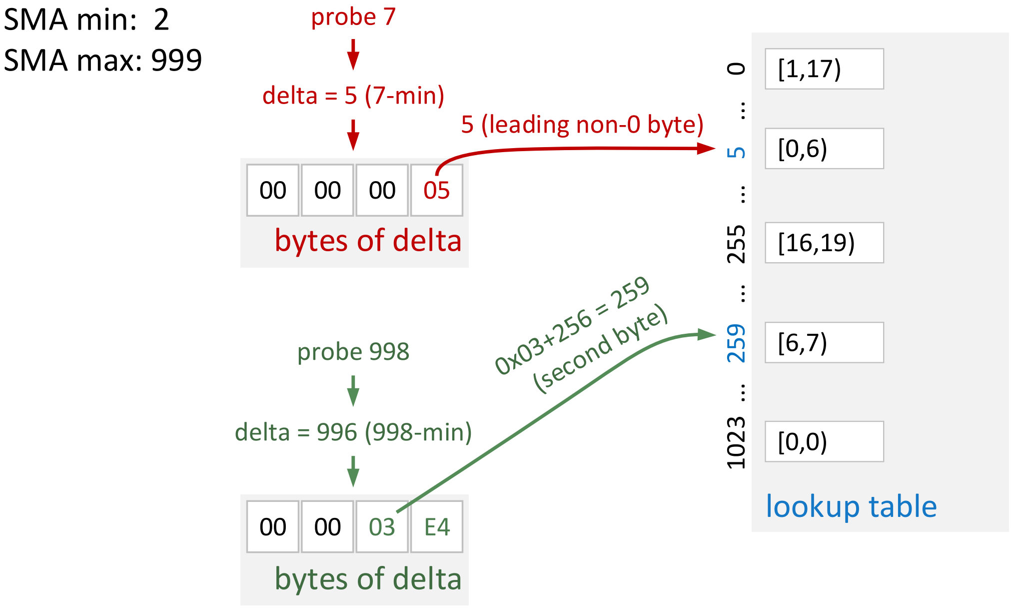


Positional SMAs - Details

- 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



Positional SMAs - Example



Challenge for JIT-compiling Query Engines

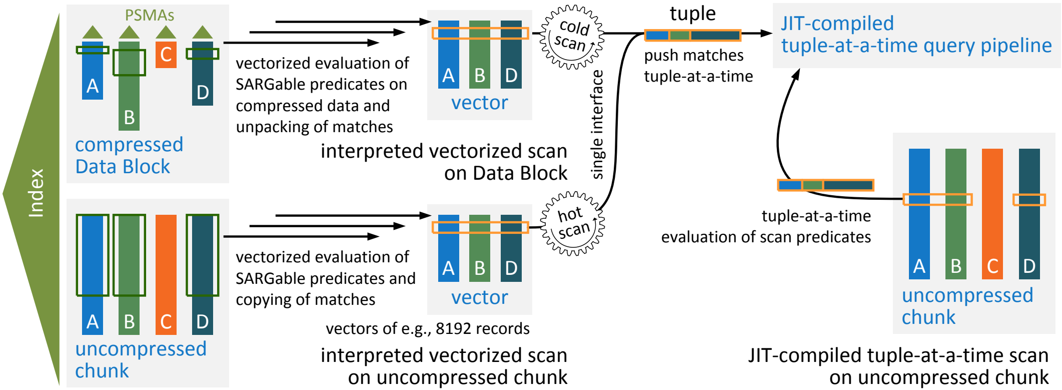
- HyPer compiles queries just-in-time (JIT) using the LLVM compiler framework
- Generated code is *data-centric* and processes a *tuple-at-a-time*

```
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

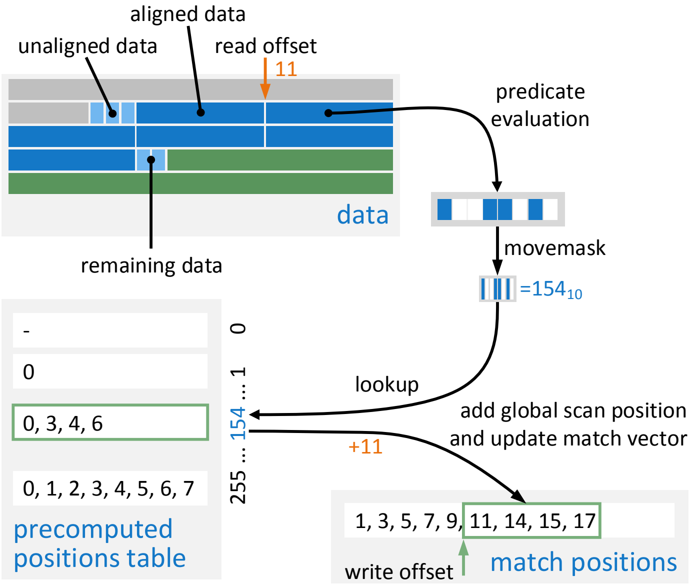
Vectorization to the Rescue

- Vectorization greatly reduces the interpretation overhead
- Spezialized 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



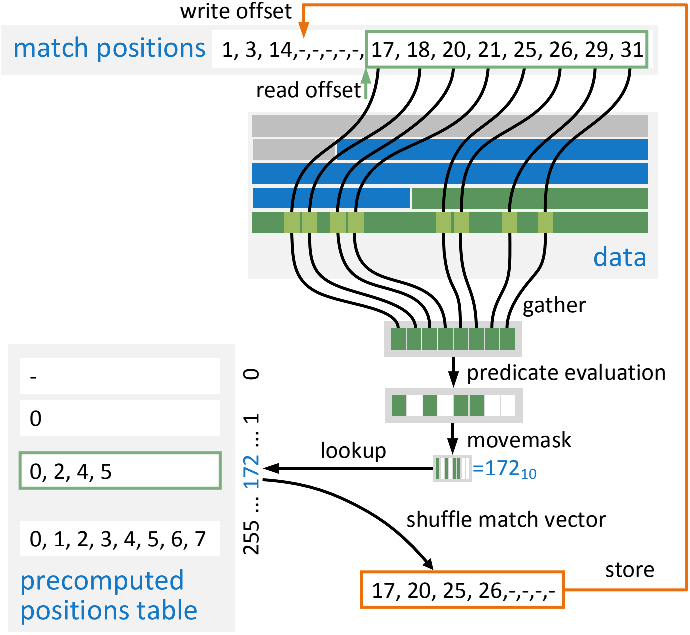
Predicate Evaluation using SIMD Instructions

Find Initial Matches



Predicate Evaluation using SIMD Instructions

Additional Restrictions



Evaluation

Compression Ratio

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

	TPC-H SF100		IMDB[1] cast info		Flights[2]	
			uncompressed			
CSV	107 GB		1.4 GB		12 GB	
HyPer	126 GB		1.8 GB		21 GB	
Vectorwise	105 GB		0.72 GB		11 GB	
			compressed			
HyPer	66 GB	(0.62×)	0.50 GB	(0.36×)	4.2 GB	(0.35×)
Vectorwise	54 GB	(0.50×)	0.24 GB	(0.17×)	3.2 GB	(0.27×)

[1] `http://www.imdb.com`

[2] `http://stat-computing.org/dataexpo/2009/`

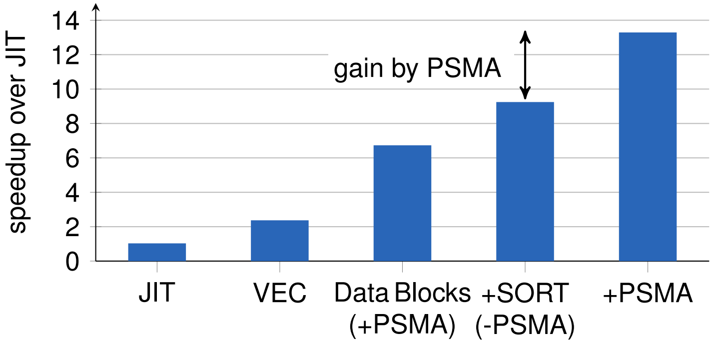
Query Performance

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

scan type	geometric mean	sum
HyPer		
JIT (uncomressed)	0.586s	21.7s
Vectorized (uncompressed)	0.583s (1.01×)	21.6s
+ SARG	0.577s (1.02×)	21.8s
Data Blocks (compressed)	0.555s (1.06×)	21.5s
+ SARG/SMA	0.466s (1.26×)	20.3s
+ PSMA	0.463s (1.27×)	20.2s
Vectorwise		
uncompressed storage	2.336s	74.4s
compressed storage	2.527s (0.92×)	78.5s

Query Performance (cont'd)

Speedup of TPC-H Q6 (scale factor 100) on block-wise sorted[3] data (+SORT).



[3]sorted by `l_shipdate`

OLTP Performance - Point Access

Throughput (in lookups per second) of random point access queries

```
select * from customer where c_custkey = randomCustKey()
```

on TPC-H scale factor 100 with a primary key index on `c_custkey`.

	Throughput [lookups/sec]	
Uncompressed	545,554	
Data Blocks	294,291	(0.54 $\times$)

OLTP Performance - TPC-C

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

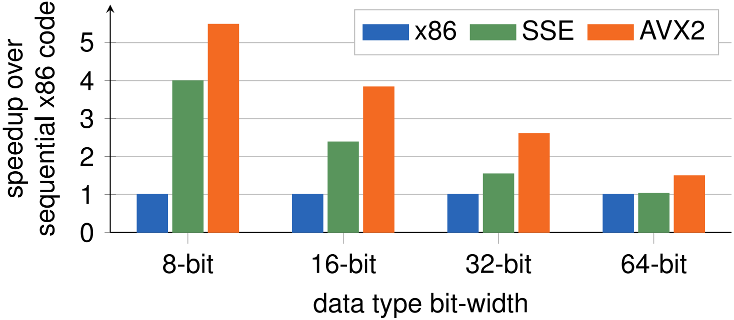
	Throughput [Tx/sec]	
Uncompressed	89,229	
Data Blocks	88,699	(0.99 ×)

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

	Throughput [Tx/sec]	
Uncompressed	119,889	
Data Blocks	109,649	(0.91 ×)

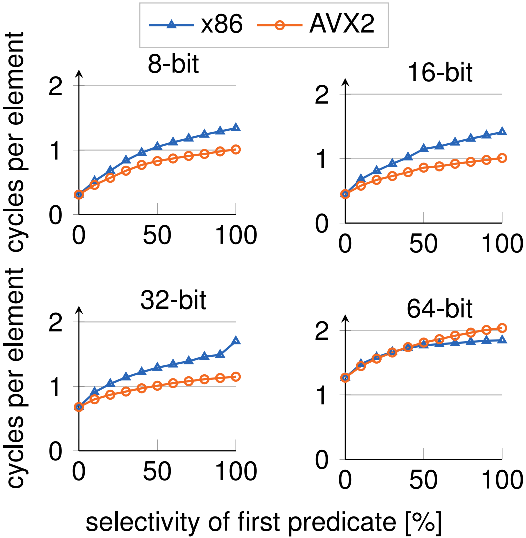
Performance of SIMD Predicate Evaluation

Speedup of SIMD predicate evaluation of type $l \leq A \leq r$ with selectivity 20%:



Performance of SIMD Predicate Evaluation (cont'd)

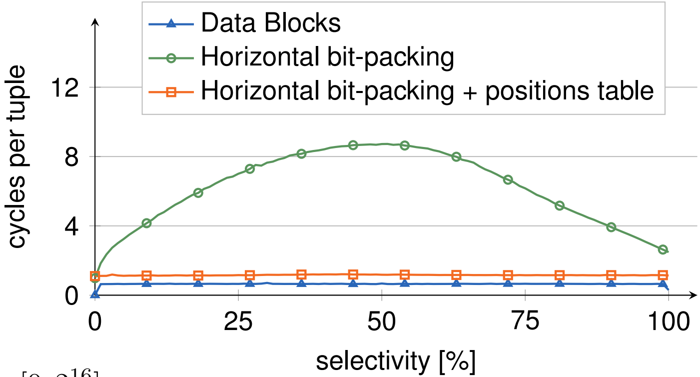
Costs of applying an additional restriction with varying selectivities of the first predicate and the selectivity of the second predicate set to 40%:



Advantages of Byte-Addressability

Predicate Evaluation

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

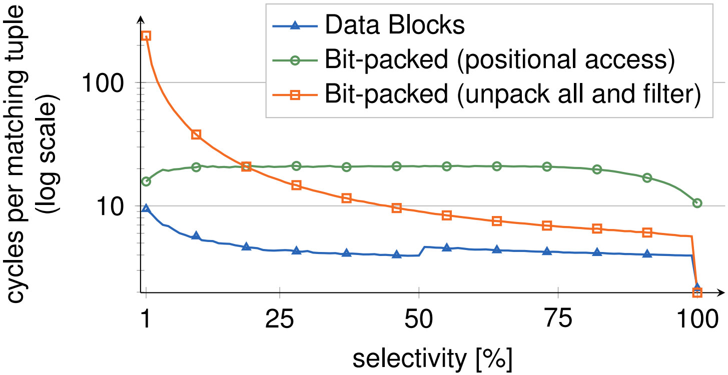


- $dom(A) = [0, 2^{16}]$
- Intentionally, the domain exceeds the 2-byte truncation by one bit
- 17-bit codes with bit-packing, 32-bit codes with Data Blocks

Advantages of Byte-Addressability

Unpacking matching tuples

Cost of unpacking matching tuples:



- 3 attributes, $dom(A) = dom(B) = [0, 2^{16}]$ and $dom(C) = [0, 2^{8}])$
- 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!