Evaluating Lightweight Integer Compression Algorithms in Column-Oriented In-Memory DBMS

Linus Heinzl    Ben Hurdelhey    Martin Boissier    Michael Perscheid    Hasso Plattner

Copenhagen, 16 August 2021
Agenda
Agenda

- Processing integers in database systems
  - Standard benchmarks
  - SAP ERP system

- Hyrise

- Integer encodings
  - Sequential and non-sequential decoding

- End-to-end benchmarks
  - TPC-DS, TPC-H, and Join Order Benchmark
  - Intel Xeon Gold 6240L, AMD EPYC 7742, and Apple M1
Processing Integers
Processing Integers

In Hyrise, decimals are stored using floats, dates as strings.
For most benchmarks, integers contribute only a small part to the footprint.

In Hyrise, decimals are stored using floats, dates as strings.
Processing Integers

For most benchmarks, integers contribute only a small part to the footprint.

At least as long as strings are present. Warehouses can be a different case.

In Hyrise, decimals are stored using floats, dates as strings.
Processing Integers

However ...

integers are highly relevant when it comes to processing time.

While join ordering is the toughest part of the JOB, most time in Hyrise goes to LIKE and IN() scans.
Processing Integers

However, integers are highly relevant when it comes to processing time. Assuming the database does not store all the data all the time in DRAM:

While join ordering is the toughest part of the JOB, most time in Hyrise goes to LIKE and IN() scans.
Processing Integers

However, integers are highly relevant when it comes to processing time. Assuming the database does not store all the data all the time in DRAM:

- most cached/buffered data in DRAM will be integer data.
Processing Integers

However, integers are highly relevant when it comes to processing time.

Assuming the database **does not store** all the data all the time in DRAM:

- most cached/buffered data in DRAM will be integer data
- even when data is tiered: DRAM is still a scarce resource
Processing Integers

Assuming the database **does not store** all the data all the time in DRAM:

- most cached/buffered data in DRAM will be integer data
- even when data is tiered: DRAM is still a scarce resource
- optimizing integers' performance/size can have a vast impact
Real-World Systems: SAP ERP
Real-World Systems: SAP ERP

What about real-world systems?
Real-World Systems: SAP ERP

What about real-world systems?

Vogelgesang et al. found that ~50% of data is stored in strings (Tableau) [1].

What about real-world systems?

Vogelsgesang et al. found that ~50% of data is stored in strings (Tableau) [1].
Real-World Systems: SAP ERP

What about real-world systems?

Vogelgesang et al. found that ~50% of data is stored in strings (Tableau) [1].

[1] Vogelgesang et al.: Get Real: How Benchmarks Fail to Represent the Real World. DBTest@SIGMOD 2018: 1-1-6
What about real-world systems?

Vogelsgesang et al. found that ~50% of data is stored in strings (Tableau) [1].

6 of the 10 most frequently accessed columns in the SAP ERP are actually integer columns.
Hyrise
Hyrise

- An columnar in-memory database
  - Speaks SQL
  - Open source
  - HTAP
  - Research & teaching
- Relevant characteristics for this talk
  - chunk-based table layout
  - intermediate query results passed via position lists

1 Hyrise on GitHub: https://git.io/hyrise
Hyrise - Storage Layout

Table T

<table>
<thead>
<tr>
<th>Chunk #1</th>
<th>Chunk #2</th>
<th>Chunk #n-1</th>
<th>Chunk #n</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segment a: dictionary-encoded</td>
<td>Segment a: dictionary-encoded</td>
<td>Segment a: unencoded</td>
<td>Segment a: unencoded</td>
</tr>
<tr>
<td>Segment b: run length-encoded</td>
<td>Segment b: dictionary-encoded</td>
<td>Segment b: unencoded</td>
<td>Segment b: unencoded</td>
</tr>
<tr>
<td>Segment c: unencoded</td>
<td>Segment c: dictionary-encoded</td>
<td>Segment c: unencoded</td>
<td>Segment c: unencoded</td>
</tr>
</tbody>
</table>


1 Hyrise on GitHub: https://git.io/hyrise
Hyrise - Storage Layout

- Every table is automatically horizontally partitioned

<table>
<thead>
<tr>
<th>Chunk #1</th>
<th>Chunk #2</th>
<th>Chunk #n-1</th>
<th>Chunk #n</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segment a</td>
<td>Segment a</td>
<td>Segment a</td>
<td>Segment a</td>
</tr>
<tr>
<td>dictionary-</td>
<td>dictionary-</td>
<td>dictionary-</td>
<td>dictionary-</td>
</tr>
<tr>
<td>encoded</td>
<td>encoded</td>
<td>encoded</td>
<td>encoded</td>
</tr>
<tr>
<td>Segment b</td>
<td>Segment b</td>
<td>Segment b</td>
<td>Segment b</td>
</tr>
<tr>
<td>run length-</td>
<td>run length-</td>
<td>run length-</td>
<td>run length-</td>
</tr>
<tr>
<td>encoded</td>
<td>encoded</td>
<td>encoded</td>
<td>encoded</td>
</tr>
<tr>
<td>Segment c</td>
<td>Segment c</td>
<td>Segment c</td>
<td>Segment c</td>
</tr>
<tr>
<td>unencoded</td>
<td>unencoded</td>
<td>unencoded</td>
<td>unencoded</td>
</tr>
</tbody>
</table>

Table T

Column T.a | Column T.b | Column T.c

---

1. Hyrise on GitHub: https://git.io/hyrise

Hyrise - Storage Layout

- Every table is automatically horizontally partitioned
  - so-called chunks

1 Hyrise on GitHub: https://git.io/hyrise

Hyrise - Storage Layout

- Every table is automatically horizontally partitioned
  - so-called chunks
- each attribute in the chunk is called a segment

---

1 Hyrise on GitHub: https://git.io/hyrise

---

Every table is automatically horizontally partitioned into so-called chunks. Each attribute in the chunk is called a segment. Once a chunk is full, it becomes immutable and a new mutable chunk is created.

1 Hyrise on GitHub: https://git.io/hyrise

Storage figure: Dreseler et al. - Hyrise Re-engineered: An Extensible Database System for Research in Relational In-Memory Data Management. EDBT 2019: 313-324
Hyrise - Storage Layout

- Every table is automatically horizontally partitioned
  - so-called **chunks**
  
- each attribute in the chunk is called a **segment**

- Once a chunk is full, it becomes **immutable** and a new mutable chunk is created

- **Insert-only**: UPDATE marks the old row as deleted and inserts updated row to the most recent chunk

---

1 Hyrise on GitHub: https://git.io/hyrise

---

Hyrise - Storage Layout

---

**Table T**

<table>
<thead>
<tr>
<th>Column $T.a$</th>
<th>Column $T.b$</th>
<th>Column $T.c$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chunk #1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Segment a</td>
<td>Segment b</td>
<td>Segment c</td>
</tr>
<tr>
<td>dictionary-encoded</td>
<td>run length-encoded</td>
<td>unencoded</td>
</tr>
<tr>
<td>Chunk #2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Segment a</td>
<td>Segment b</td>
<td>Segment c</td>
</tr>
<tr>
<td>dictionary-encoded</td>
<td>dictionary-encoded</td>
<td>dictionary-encoded</td>
</tr>
<tr>
<td>Chunk #n-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Segment a</td>
<td>Segment b</td>
<td>Segment c</td>
</tr>
<tr>
<td>unencoded</td>
<td>unencoded</td>
<td>unencoded</td>
</tr>
<tr>
<td>Chunk #n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Segment a</td>
<td>Segment b</td>
<td>Segment c</td>
</tr>
<tr>
<td>unencoded</td>
<td>unencoded</td>
<td>unencoded</td>
</tr>
</tbody>
</table>

1. Hyrise on GitHub: https://git.io/hyrise

---

Hyrise - Storage Layout

- Segments can be **encoded independently**

---

Table T

<table>
<thead>
<tr>
<th>Chunk #1</th>
<th>Column T.a</th>
<th>Column T.b</th>
<th>Column T.c</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segment a</td>
<td>dictionary-encoded</td>
<td>run length-encoded</td>
<td>unencoded</td>
</tr>
<tr>
<td>Segment b</td>
<td>dictionary-encoded</td>
<td>run length-encoded</td>
<td>unencoded</td>
</tr>
<tr>
<td>Segment c</td>
<td>dictionary-encoded</td>
<td>run length-encoded</td>
<td>unencoded</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chunk #2</th>
<th>Column T.a</th>
<th>Column T.b</th>
<th>Column T.c</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segment a</td>
<td>dictionary-encoded</td>
<td>dictionary-encoded</td>
<td>dictionary-encoded</td>
</tr>
<tr>
<td>Segment b</td>
<td>dictionary-encoded</td>
<td>dictionary-encoded</td>
<td>dictionary-encoded</td>
</tr>
<tr>
<td>Segment c</td>
<td>dictionary-encoded</td>
<td>dictionary-encoded</td>
<td>dictionary-encoded</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chunk #n-1</th>
<th>Column T.a</th>
<th>Column T.b</th>
<th>Column T.c</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segment a</td>
<td>unencoded</td>
<td>unencoded</td>
<td>unencoded</td>
</tr>
<tr>
<td>Segment b</td>
<td>unencoded</td>
<td>unencoded</td>
<td>unencoded</td>
</tr>
<tr>
<td>Segment c</td>
<td>unencoded</td>
<td>unencoded</td>
<td>unencoded</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chunk #n</th>
<th>Column T.a</th>
<th>Column T.b</th>
<th>Column T.c</th>
</tr>
</thead>
<tbody>
<tr>
<td>Segment a</td>
<td>unencoded</td>
<td>unencoded</td>
<td>unencoded</td>
</tr>
<tr>
<td>Segment b</td>
<td>unencoded</td>
<td>unencoded</td>
<td>unencoded</td>
</tr>
<tr>
<td>Segment c</td>
<td>unencoded</td>
<td>unencoded</td>
<td>unencoded</td>
</tr>
</tbody>
</table>

---


---

1 Hyrise on GitHub: https://git.io/hyrise
Hyrise - Storage Layout

- Segments can be **encoded independently**
- On top of the segment encoding type, internal integer vectors can be further compressed

Table T

<table>
<thead>
<tr>
<th>Column T.a</th>
<th>Column T.b</th>
<th>Column T.c</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chunk #1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Segment a</td>
<td>Segment b</td>
<td>Segment c</td>
</tr>
<tr>
<td>dictionary-encoded</td>
<td>run length-encoded</td>
<td>unencoded</td>
</tr>
<tr>
<td>Chunk #2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Segment a</td>
<td>Segment b</td>
<td>Segment c</td>
</tr>
<tr>
<td>dictionary-encoded</td>
<td>dictionary-encoded</td>
<td>dictionary-encoded</td>
</tr>
<tr>
<td>Chunk #n-1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Segment a</td>
<td>Segment b</td>
<td>Segment c</td>
</tr>
<tr>
<td>unencoded</td>
<td>unencoded</td>
<td>unencoded</td>
</tr>
<tr>
<td>Chunk #n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Segment a</td>
<td>Segment b</td>
<td>Segment c</td>
</tr>
</tbody>
</table>
| unencoded   | unencoded  | unencoded  


1 Hyrise on GitHub: https://git.io/hyrise
Hyrise - Storage Layout

- Segments can be **encoded independently**
- On top of the segment encoding type, internal integer vectors can be further compressed
  - e.g., the offset vector of a dictionary encoded segment

---

1. Hyrise on GitHub: https://git.io/hyrise

Hyrise - Storage Layout

- Segments can be **encoded independently**
- On top of the segment encoding type, internal integer vectors can be further compressed
  - e.g., the offset vector of a dictionary encoded segment
  - storing offsets as `uint8_t` for small dictionaries

1. Hyrise on GitHub: https://git.io/hyrise
Hyrise - Intermediate Results

- Hyrise tries to materialize results as late as possible
- Intermediate results are passed to following operators in form of **position lists**: 
  - store pairs of `<chunk_id, row_id>` that reference the initial data table
- Effect:
  - small intermediates are passed (small payloads for joins etc.)
  - potentially expensive materialization when position lists is scrambled (e.g. after a join)

---

1 Hyrise on GitHub: https://git.io/hyrise
Integer Encodings
Integer Encoding Libraries
Integer Encoding Libraries

TurboPFor: Fastest Integer Compression

- TurboPFor: The new synonym for "integer compression"
  - (2019.11) ALL functions now available for 64 bits ARMv8 NEON & Power9 Altivec
Integer Encoding Libraries

TurboPFor: Fastest Integer Compression

- TurboPFor: The new synonym for "integer compression"
  - (2019.11) ALL functions now available for 64 bits ARMv8 NEON & Power9 Altivec

The FastPFOR C++ library: Fast integer compression
Integer Encoding Libraries

TurboPFor: Fastest Integer Compression

- TurboPFor: The new synonym for "integer compression"
  - (2019.11) ALL functions now available for 64 bits ARMv8 NEON & Power9 Altivec

streamvbyte

The FastPFOR C++ library: Fast integer compression
Integer Encoding Libraries

TurboPFOR: Fastest Integer Compression
- TurboPFOR: The new synonym for "integer compression"
  - (2019.11) ALL functions now available for 64 bits ARMv8 NEON & Power9 Altivec

streamvbyte
- Ubuntu 20.04 CI (GCC 9) passing

SIMDCompressionAndIntersection
- build passing

MaskedVByte
- build passing

Oroch
- A C++ library for integer array compression.

The FastPFOR C++ library: Fast integer compression
- build passing
- build testing

dictionary
- High-performance dictionary coding

Compact vector
- This library provide a bit-packed vector like data structure for storing integer types.
Integer Encoding Libraries

- From libraries stem from the field of information retrieval
  - optimized for set operations
  - optimized for sequential operations
- Only few support efficient random access to single positions
  - relevant for Hyrise's processing of position lists
### Table 1: Open-source Lightweight Integer Encoding Implementations

<table>
<thead>
<tr>
<th>Name</th>
<th>License</th>
<th>Technique</th>
<th>Language</th>
<th>Random Access</th>
<th>Documented</th>
<th>Tested &amp; Matured</th>
<th>Platforms</th>
</tr>
</thead>
<tbody>
<tr>
<td>FastPFfor</td>
<td>Apache 2.0</td>
<td>FoR</td>
<td>C++</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>x64/SSE3</td>
</tr>
<tr>
<td>TurboPFfor</td>
<td>GPL</td>
<td>FoR, NS</td>
<td>C</td>
<td>●</td>
<td>○</td>
<td>●</td>
<td>x64/AVX2, ARM/NEON, Power9</td>
</tr>
<tr>
<td>SIMDCompressionAnd-Intersection (SIMDCAI)</td>
<td>Apache 2.0</td>
<td>FoR, NS</td>
<td>C++</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>x64/SSE4.1</td>
</tr>
<tr>
<td>MaskedVByte</td>
<td>Apache 2.0</td>
<td>NS</td>
<td>C</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>x64/SSE4.1</td>
</tr>
<tr>
<td>StreamVByte</td>
<td>Apache 2.0</td>
<td>NS</td>
<td>C</td>
<td>○</td>
<td>●</td>
<td>●</td>
<td>x64/SSE4, ARM</td>
</tr>
<tr>
<td>Oroch</td>
<td>MIT</td>
<td>FoR, NS</td>
<td>C++</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>cross-platform</td>
</tr>
<tr>
<td>dictionary</td>
<td>Apache 2.0</td>
<td>DICT</td>
<td>C++</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>x64/AVX512</td>
</tr>
<tr>
<td>compact_vector</td>
<td>MIT</td>
<td>NS</td>
<td>C++</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>cross-platform</td>
</tr>
</tbody>
</table>

Table 1: Open-source Lightweight Integer Encoding Implementations
Table 2: CPU time in microseconds for decoding the whole vector, per library. For sequential decoding, runtime measured as mean across all data distributions. Compression rate as average across data distributions.
End-to-End Evaluation

We only show the vector compression evaluations and skip the segment encoding analysis in this presentation. See the paper for all analyses.

TPC-H and DS use a scale factor of 10
Multi-Threaded: slightly more concurrent clients than threads to maximize load
TPC-H - Vector Compression

Intel Xeon Gold 6240L

AMD EPYC 7742
TPC-H - Performance Counters

- bitpacking_compactvector
- bitpacking_compactvector_16
- bitpacking_for_SIMDCAI_simd_simd
- bitpacking_turboPFOR
- bytepacking_Hyrise_fsba
- std::vector<uint32_t>

Cache misses (in million, measured using Intel PCM)

L2 Misses
L3 Misses

Size integer columns [GB]

Memory bandwidth [GB/s]
Evaluating Lightweight Integer Compression Algorithms in Column-Oriented In-Memory DBMS

Linus Heinzl
Hasso Plattner Institute
University of Potsdam, Germany
linus.heinzl@student.hpi.de

Ben Hurdelhey
Hasso Plattner Institute
University of Potsdam, Germany
ben.hurdelhey@student.hpi.de

Martin Boissier
Hasso Plattner Institute
University of Potsdam, Germany
martin.boissier@hpi.de

Michael Perscheid
Hasso Plattner Institute
University of Potsdam, Germany
michael.perscheid@hpi.de

Hasso Plattner
Hasso Plattner Institute
University of Potsdam, Germany
hasso.plattner@hpi.de

ABSTRACT
Lightweight data compression algorithms are often used to decrease memory consumption of in-memory databases. In recent years, various integer compression techniques have been proposed that focus on sequential encoding and decoding and exploit modern CPUs' vectorization capabilities. Interestingly, another dominant access pattern in databases systems has seen little attention: random access decoding. In this paper, we compare end-to-end database performance for various integer compression codecs on three recent CPU architectures. Our evaluation suggests that random access performance is often more relevant than vectorization capabilities for sequential accesses. Before integrating selected encodings in the database core, we benchmarked seven libraries in an exhaustive standalone comparison. We integrated the most promising techniques into the relational in-memory database system Hyrise and evaluated their performance for TPC-H, TPC-DS, and the Join Order Benchmark on three different CPU architectures. Our results emphasize the importance of random access decoding. Compared to state-of-the-art dictionary encoding in TPC-H, alternatives allow reducing memory consumption of integer columns by up to 53% while improving runtime performance by 5% on an Intel CPU and over 16% on an Apple M1.

1 COMPRESSION IN IN-MEMORY DATABASE SYSTEMS
With increasing volumes of data being collected, the need for fast and efficient processing of workloads increases continuously. When customers' workloads have substantial performance requirements that the Database Management System (DBMS) needs to serve, storing data on disk is often too slow. Technological advances in the main memory industry have made large main memory capacities affordable, enabling the adoption of in-memory databases [17]. For these DBMS, disk access is no longer the bottleneck. Instead, main memory access and processing efficiency are the new optimization goals [29]. Therefore, customers often choose to use in-memory databases to serve their performance-critical workloads. However, for in-memory databases, a compact representation of data is even more crucial. In a recent survey by the market researcher IDG [19], 80% of the participating organizations stated that they have at least one part of their infrastructure running in the cloud and spend around one-third of their IT budget on cloud computing. Here, the bill depends on the used resources, which is often measured in RAM size. To adapt in-memory databases to run efficiently on cloud infrastructures, therefore, means that they should use as few main memory as possible. Data compression can help to achieve this. However, several practical considerations limit the usage of compression. In addition to the strong performance requirements, Service Level Agreements (SLAs) in enterprise contexts often bound the maximum allowed query response time [30], which makes the usage of heavyweight compression schemes (e.g., LZ4 [9], LZ77 [38]) difficult. As an alternative, lightweight encoding schemes can still be used.

Figure 1: TPC-H runtimes and sizes for various integer encoding schemes in Hyrise (single-threaded, SF 10, sum of mean query runtimes). On all three evaluated platforms, bit-packing efficiently balances runtime performance and memory consumption while std::vector is the fastest alternative in the single-threaded setting.

For AWS EC2 instances, costs correlate to DRAM sizes: https://aws.amazon.com/de/ec2/pricing/on-demand/
Lessons Learned
Lessons Learned

- Integers often contribute only a small share to the overall memory consumption
  - different for processing time
  - even more so, when data is tiered/buffered
Lessons Learned

- Integers often contribute only a small share to the overall memory consumption
  - different for processing time
  - even more so, when data is tiered-buffered
- Balancing performance and memory footprint is tough
  - single-threaded performance can differ significantly from multi-threaded
  - highly depends on the load (50% load is more comparable to ST than MT)
Lessons Learned

- Integers often contribute only a small share to the overall memory consumption
  - different for processing time
  - even more so, when data is tiered/buffered

- Balancing performance and memory footprint is tough
  - single-threaded performance can differ significantly from multi-threaded
  - highly depends on the load (50% load is more comparable to ST than MT)

- Hard to generalize performance insights
  - Calibration/learned prediction models needed for selection
Lessons Learned

- Integers often contribute only a small share to the overall memory consumption
- different for processing time
- even more so, when data is tiered/buffered
- Balancing performance and memory footprint is tough
  - single-threaded performance can differ significantly from multi-threaded
  - highly depends on the load (50% load is more comparable to ST than MT)
- Hard to generalize performance insights
  - Calibration/learned prediction models needed for selection