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[1]. 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

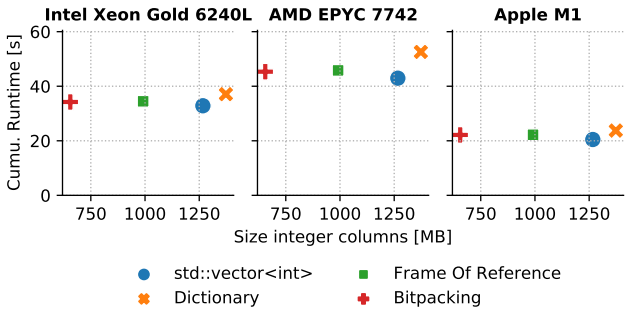


Figure 1: TPC-H runtimes and sizes for various integer encoding schemes in Hyrise (single-threaded, SF 10, sum of mean query runtimes). On all three evaluated platforms, bitpacking efficiently balances runtime performance and memory consumption while `std::vector` is the fastest alternative in the single-threaded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

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

provide reasonable compression rates while offering fast decoding speeds [10, 25, 32]. Figure 1 shows the single-threaded runtime performance and cumulative size of all integer columns when being encoded with different integer encoding schemes in Hyrise (TPC-H scale factor 10; runtime is the sum of the mean runtimes of all 22 queries). The widely used state of the art Dictionary Encoding performs significantly worse than alternatives such as bitpacking or not encoding data at all, while also consuming more memory. While the Apple M1 is the fastest CPU in this setting (cf. Section 4.2 for remarks on comparability), the relative performance differences are stable on all three platforms. In the later evaluation, we will analyze if these results hold for multi-threaded executions.

We focus on evaluating lightweight, lossless, integer compression schemes. These schemes have made notable advances in recent years through the use of vectorization [10, 26, 27, 36]. Additionally, as we explain in Section 2, integer compression significantly reduces memory consumption for various workloads. While integer compression schemes have shown to be performant in standalone benchmarks, their effect on end-to-end DBMS performance is more nuanced. With this paper, we are going to make the following contributions:

- An analysis of data types and access frequencies in benchmarks as well as in the real world in Section 2, which leads to our focus on integer compression.
- We examine frequent access patterns of an in-memory DBMS. Furthermore, we analyze the difficulty of leveraging recent compression vectorization trends to DBMS applications that require efficient random accesses in Section 3.
- A standalone comparison of seven open-source integer compression libraries where we compare memory consumption and performance for different codecs, access patterns, and data distributions in Section 4.
- We compare state-of-the-art encoding schemes in an end-to-end fashion in the columnar in-memory research DBMS Hyrise. We show that several lightweight integer encoding schemes offer performance improvements while at the same time providing compression benefits over currently used schemes such as Dictionary Encoding.

We use the term *encoding* to describe encoding a column segment of nullable table data, while *compression* describes second-order compression of an auxiliary integer vector. In addition, *codec* and *scheme* describe the implementation of a *compression algorithm*.

2 SYNTHETIC AND REAL-WORLD DATA CHARACTERISTICS

We analyzed data and workloads of TPC-H, TPC-DS, and Join Order Benchmark (JOB) to highlight the importance of integer columns in analytical workloads.

Their role may not be obvious at first sight. As Figure 2 shows, the prevalence of data types varies visibly between the benchmarks, with integer columns not being dominant in either one[2,3]. However, their importance becomes much clearer when we consider the runtime of the benchmarks. We see that the analytical workloads

[2]Hyrise stores decimals as floats and datetimes as strings.
[3]In cases where, e.g., an aggregate groups by a single column and calculates an aggregate on another column, we split the runtime equally between both columns.

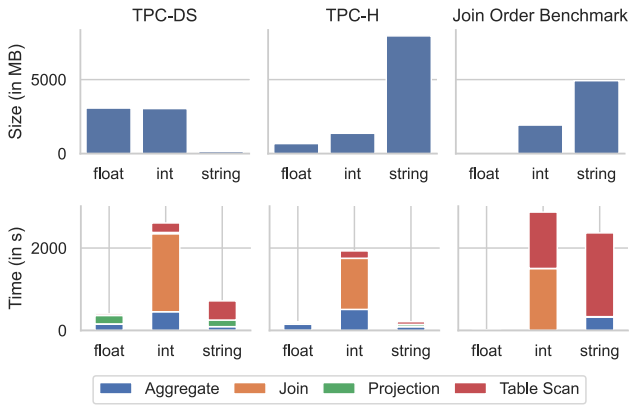


Figure 2: Overview of data type sizes and runtimes for TPC-H (SF 10), DS (SF 5), and Join Order Benchmark (JOB).

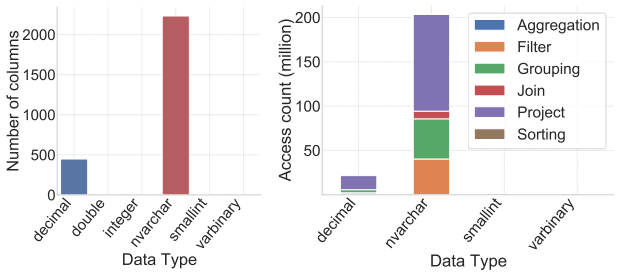
spend between 25 % and 50 % of the time on joins, which operate exclusively on (integer) key columns. The analytical workloads spend more time on integer columns in each benchmark than on all other combined columns.

For an in-memory database such as Hyrise [15], we can leverage this insight to optimize performance and memory usage. The idea here is to store seldom accessed columns on slower storage tiers such as Non-volatile Memory (NVM), SSD, or disk [34]. We then find ourselves with a large part of the in-memory storage consisting of the frequently accessed integer columns, allowing significant improvements through compression.

2.1 Real-World ERP Data Analysis

We wanted to determine whether the integer-heavy patterns found in the benchmarks also exist in real-world data or if string-heavy patterns, similar to the ones that Vogelsgesang et al. [35] found, prevail. Therefore, we investigated the workload and data of an S4/HANA ERP production system of a Fortune 500 company.

Figure 3a shows the number of columns in the analyzed system per data type. We found that over 83 % of all columns are string columns. This value is even higher than the reported values of the analyzed Tableau workbooks by Vogelsgesang et al.



(a) Number of columns by data type.

(b) Number of column accesses per data type.

Figure 3: System analysis of a large SAP S4/HANA customer.

To analyze how often which columns are accessed, we analyzed the query plan cache of the system. Figure 3b shows the cumulative

results. Again, string columns account for the largest share. The high number of projections is caused by regular exports to the internal data warehouse system. These exports extract new tuples – which often have hundreds of attributes (cf. [3, 5]) – in their full width. But even when projections are ignored, the vast majority of accesses is on string columns. Of the ten most frequently accessed columns, all ten are string columns.

One of the main reasons for the dominance of string columns in this S4/HANA system is ensuring compatibility with third-party software and legacy systems. By using strings, old systems that did not enforce numeric columns can be upgraded without rewriting data. Moreover, the use of strings for numeric columns increases flexibility for customers. Of the ten most frequently accessed columns, six columns contain only numeric data. Even though this customer uses numeric values (and in fact, the column names even denote that this string column is actually a numeric one), the ERP system's default is to store these columns as string columns. As a consequence, we consider integer column optimizations to be also of value for enterprise systems, given this high percentage of frequently accessed actually being integer columns.

Moreover, as we discuss in Section 5, systems such as Hyrise or SAP HANA also use integer vectors for several internal applications, such as Dictionary Encoding, which is another reason we decided to focus on integer compression.

3 INTEGER COMPRESSION ALGORITHMS AND IMPLEMENTATIONS

The main goal of this paper is to evaluate integer compression schemes that both improve performance and compression rates in DBMS. Before we discuss algorithms and implementations for this, we discuss and investigate access patterns that they must support: sequential and random access decoding.

3.1 Access Patterns for Compressed Data

This section explains why we require sequential and random access patterns and investigate how often they occur. To understand where the different access patterns originate from, we have to understand how DBMS pass intermediate results between query operators that compose the operator tree (cf. [15]).

3.1.1 Column-oriented Representations of Intermediate Results. We consider two strategies for representing intermediate results between operators: position lists and intermediate materialization. We distinguish this definition from the terms early materialization and late materialization, which describe the point of time when the DBMS constructs tuples from column-oriented data [2].

The first strategy for passing intermediate data is the usage of Position Lists (PosLists). It is extensively used by Hyrise [15], SAP HANA [16], and partially MonetDB [20]. These PosLists offer constrained data characteristics (i.e., as they are used as indices whose integers are unsigned and cannot be larger than the initial input tables) and avoid materializing and passing the actual data between operators. A drawback of the PosList approach is that duplicate work must be done when accessing data more than once (e.g., multiple joins on the same column). Furthermore, in cases such as expression evaluations (e.g., `a + 4`), merely passing a PosList is impossible due to the alteration of underlying data.

The second strategy, materialization of intermediate results, is implemented by systems such as MonetDB [20], MorphStore [12], and Vertica [23].

Notably, also systems that pass materialized intermediate results require efficient non-sequential access to columns, e.g., when materializing the projected columns after a scan operation. Hyrise delays the materialization as much as possible, ideally until the result is returned to the user. This approach significantly reduces the data that is passed between operators. However, it can also add many random accesses when tuples need to be reconstructed. This reconstruction can be on an ordered position list (e.g., returning a table that has only been filtered). But it can also happen on a position list that is more or less in random order (e.g., the result of a radix-clustered inner join, cf. [14]).

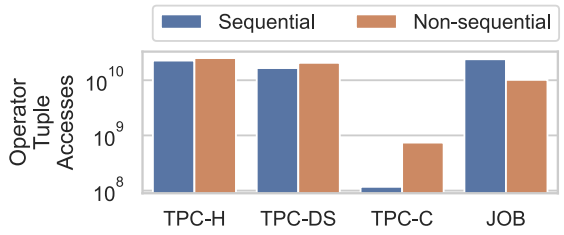


Figure 4: Tuple access count by access type of all operators

3.1.2 Random Access through Position Lists. We will focus on the first strategy, passing intermediate results through positional references, such as implemented in Hyrise [15]. Let us consider a simple aggregated sum over a column. As input, the operator receives the encoded base data and the position list of indices from the previous operator's output. This PosList can be the output of a predicate and thus reference substantially less data than the referenced segment holds. Subsequent predicates and key-selecting Online Transaction Processing (OLTP) workloads such as implemented in the TPC-C benchmark only amplify this effect. Furthermore, the PosList does not have to be sorted, as join operators shuffle the indices.

The first option for computing the sum is to fully decode and materialize the underlying data and then access it at the indices from the PosList. If the PosList only contains very few references while the original encoded segment has much more entries, this can be inefficient. Depending on the context, it could make more sense to use random access decoding and only retrieve the desired indices' data.

The workload analyses shown in Figure 4 indicate that the second option with random access is worth considering: on average, more than 44 % of all tuple accesses by operators can be performed through PosList access on other operators' output. Furthermore, the mean selectivity per predicate is 36 %, implying the presence of significantly shorter PosLists.

3.2 Compression Algorithms

For our goal to optimize memory consumption and performance, we consider four lightweight integer compression techniques [10] concerning random access.

Name	License	Technique	Language	Random Access	Documented	Tested & Matured	Platforms
FastPFor	Apache 2.0	FoR	C++	○	●	●	x64/SSE3
TurboPFor	GPL	FoR, NS	C	◐	○	●	x64/AVX2, ARM/NEON, Power9
SIMDCompressionAnd-Intersection (SIMDCAI)	Apache 2.0	FoR, NS	C++	◐	●	●	x64/SSE4.1
MaskedVByte	Apache 2.0	NS	C	●	●	●	x64/SSE4.1
StreamVByte	Apache 2.0	NS	C	○	●	●	x64/SSE4, ARM
Oroch	MIT	FoR, NS	C++	●	●	●	cross-platform
dictionary	Apache 2.0	DICT	C++	●	●	○	x64/AVX512
compact_vector	MIT	NS	C++	●	●	●	cross-platform

Table 1: Open-source Lightweight Integer Encoding Implementations

Random Access	<technique>_<library> [_<specialization>]	Sequential Decoding	PosList Decoding by Length				Compression Rate
			1	10^3	10^4	10^6	
●	*for_SIMDCAI_simd*	10.1	0.007	6.0	60.1	6044	0.26
	pfor_turboPFOR	54.3	0.006	5.0	55.1	5494	0.34
	bitpacking_turboPFOR	29.0	0.007	2.5	40.0	3053	0.38
	bitpacking_compactvector	161.3	0.005	3.0	32.7	6976	0.39
	bitpacking_Oroch_intArray	14 820.3	0.29	245.4	2308.0	204 491	0.57
	Unencoded (std::vector<T>)	9.3	0.002	0.8	15.5	1470	1.0
○	*pfor_fastPFOR*	10.7	10.5	11.3	16.2	854	0.34
	bytepacking_MaskedVByte	87.1	124.7	73.9	87.5	1742	0.57
	bytepacking_StreamVByte	17.4	14.8	20.9	22.4	1991	0.60

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.

- Delta encoding works by only storing the difference to the previous value [33]. Because we require knowledge of previous entries, this technique does not provide constant-time random access decoding.
- Frame of Reference (FoR) represents each original value as a difference to a reference value [18]. The Patched Frame of Reference scheme [26] is an extension to it, and we expect both to offer fast random access.
- Dictionary Encoding [1] stores all distinct values in a sorted dictionary and represents the original sequence (Attribute Vector (AV)) through offsets into the dictionary..
- Null suppression (NS) [1] is implemented in VByte [27] or varint [13]. It either operates bit- (*bitpacking*) or byte-aligned (*bytepacking*). We can apply it independently per value, or uniformly with fixed width. We focus on the latter, as only this allows fast random access.

As a reference, we further evaluate Hyrise's *Unencoded* segment encoding in the end-to-end benchmarks, which stores its segment data without any compression in a plain `std::vector<T>` (plus an optional `std::vector<bool>` to mark NULL values).

3.2.1 Lightweight Integer Compression Libraries. For each compression technique, we use open-source implementations shown in Table 1. Since this work aims to improve encoding size and performance in an actual DBMS, we do not consider libraries that are not well tested or no longer actively maintained. We experienced many memory access violations during the integration of the *TurboPFOR* library and did not find sufficient documentation on input constraints.

Furthermore, we observe a mismatch between recent compression algorithm research and the demand for random access decoding in our DBMS application Hyrise. While DBMS alongside search engines are a frequently referenced application of compression [25, 26, 36], random access decoding is often not or only poorly supported by these libraries.

4 EVALUATION OF INTEGER SEGMENT ENCODING ALGORITHMS AND IMPLEMENTATIONS

To evaluate the libraries' schemes from Table 1, we proceed in two steps. First, we create an exhaustive set of benchmarks for all

codecs that the libraries offer. There, we model the access patterns described in Section 3. In the second step, we then implement and benchmark only the most promising candidates in Hyrise to reduce implementation efforts.

4.1 Standalone Evaluation on Synthetic Data

To evaluate the selection of codecs from Table 1, we first create simple standalone benchmarks that approximate the access patterns described in Section 3. In the benchmarked scenario, we operate on a vector C of integer values. We set its size $|C|$ to the default chunk size in Hyrise, $2^{16} - 1$. Per benchmark, we fill C with synthetic sequences of integers and compress it using the compression scheme under test. We then measure the required CPU time for sequential decoding and random access decoding. In Hyrise, sequential encoding performance is of lower importance. While the system could re-encode the immutable segments (cf. [15]), it typically encodes segments only once when the most recent chunk reaches its tuple limit and becomes immutable. The synthetic benchmarks have been executed on an Intel Xeon Platinum 8180 CPU.

4.1.1 Sequential Decoding. To test the sequential decoding performance of the libraries, we create sequences of integers that model several distributions: (i) uniformly distributed random numbers between 0 and 2^{32}, (ii) sorted numbers with an equal distance of 5, (iii) years between 1900 and 2100, (iv) months between 1 and 12, (v) a time series starting at 10^6.

One application of sequential decoding is the linear table scan that directly operates on base table data (i.e., no PosList is given). For them, it can be advantageous to decode the whole segment all at once because this allows vectorization. Among the compression schemes, only *bitpacking_Oroch_intArray* and *bitpacking_compactvector* do not leverage vectorized sequential decoding, which is apparent in the results in Table 2. Here, both of them are orders of magnitudes slower than the fastest schemes, *for_SIMDCAI_simd* and *pfor_fastPFOR*. The compression rate ranges from 0.26 for *for_SIMDCAI_simd* to 0.6 for *bytepacking_StreamVByte*.

4.1.2 Random Access. As we have explained in Section 3, Hyrise passes intermediate query results via positional references. Depending on data and workload, the size of an intermediate PosList varies. To test how the PosList size affects the decoding performance, we create PosLists of different sizes, ranging from 1 to 10^7 elements. Each PosList contains a random set of indices in $[0, |C|)$, referencing a vector C of uniformly distributed values. The task is to decode all referenced entries.

The schemes that support random access decoding only access the indices specified in the PosList. For the other schemes, we first sequentially decode the whole vector and then retrieve the values at the indices given in the PosList.

As we can see in Table 2, random access allows faster decoding speed for short PosLists. The point where vectorized sequential decoding (if possible) performs better than the respective random access implementation varies between schemes. For *bitpacking_turboPFOR*, the point lies just short of a PosList length of 10^4, which equals $\frac{1}{6}$ of the chunk size. Above this, sequential decoding performs better due to optimizations and the use of vectorization. In Hyrise, the default length of the PosList is $2^{16} - 1$, but as discussed, predicates quickly decrease this further, which is why we focus on schemes that support random access decoding.

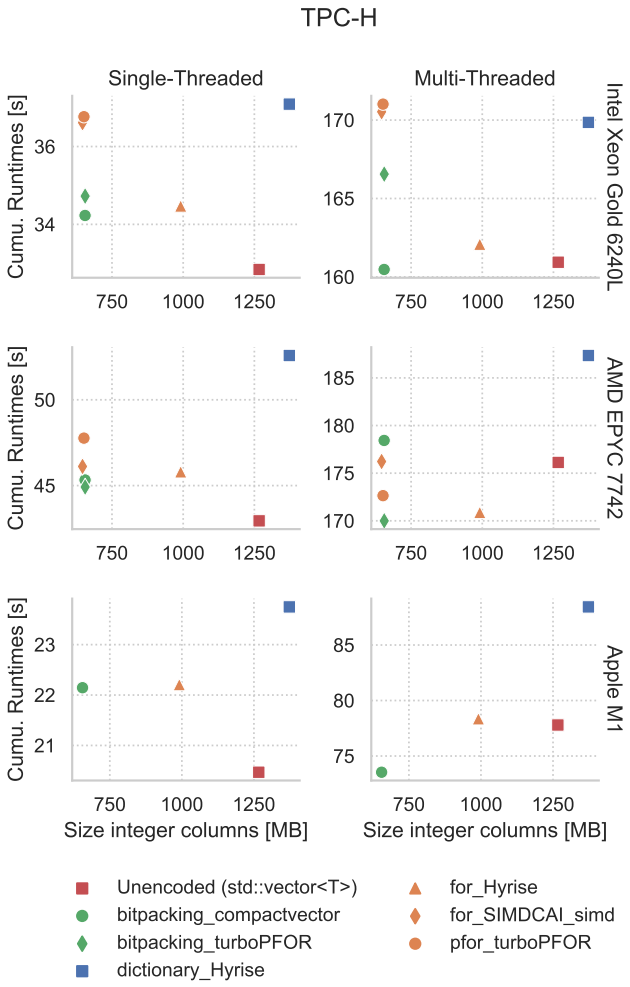
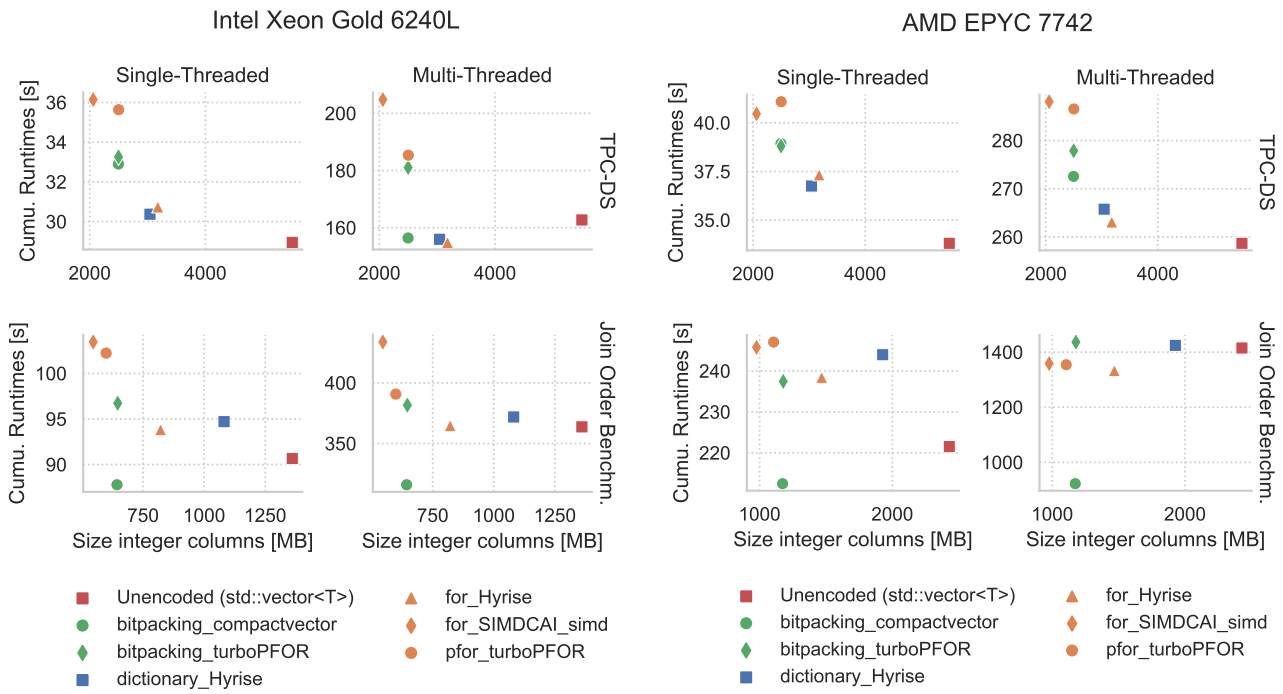


Figure 5: Size of integer columns and runtime performance of various segment encodings (TPC-H SF 10) for the three evaluated CPU architectures, single- and multi-threaded. Not all libraries can be built on an Apple M1.

4.2 End-to-End Evaluation Setup

For the end-to-end benchmarks, we evaluated various benchmarks on three CPU architectures: **(i)** an Intel Xeon Gold 6240L (18 physical cores, 36 threads, 1024 KB L2 cache, 24.75 MB shared L3 cache), **(ii)** an AMD EPYC 7742 (64 physical cores, 128 threads, 512 KB L2 cache, 256 MB shared L3 cache), and **(iii)** an Apple M1 (2020 M1 Mac Mini, 4 high-performance cores, 4 high-efficiency cores, 320/192 KB L1 cache, 12 MB shared L2 cache). For measurements on multi-socket systems, we pinned processes and memory to a single node using `numactl` to exclude any NUMA effects. The operating systems version are Ubuntu 20.10 (GNU/Linux 5.9.10) and MacOS 11.4 with clang-12 as the compiler. Per benchmark, we specify whether we run it in single-threaded or multi-threaded mode.



(a) Intel Xeon Gold 6240L

(b) AMD EPYC 7742

Figure 6: Size of integer columns and runtime performance of various integer segment encodings (TPC-DS SF 10 and Join Order Benchmark), single- and multi-threaded.

For the latter, we spawn multiple clients that issue the benchmark queries in a randomized order. The number of clients is set in way to maximize the load on the system. For a CPU with t threads, we spawn $\lceil max(t+1, t*1.1) \rceil$ clients (i.e., 140 clients for the 64 cores and 128 threads of the AMD EPYC 7742 CPU, 40 clients for the Intel Xeon Gold 6240L, and 9 clients for the Apple M1). We do not recommend running that many concurrent clients for Hyrise and using these settings solely to maximize load on the CPU. The Hyrise instance is set to use all physical and logical cores.

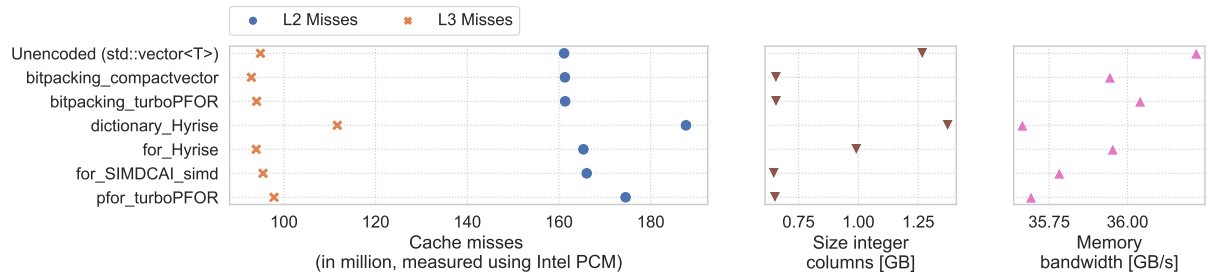
Please note that the shown performance results for different CPU architectures are not meant to compare them with each other but rather to compare the relative performance between various compression schemes on different platforms. The single-threaded benchmark shown in Figure 1 uses a single CPU core (of up to 64 of the AMD EPYC 7742) and for each CPU the clock speed when using a single core only (which we do not consider to be a realistic scenario) can vastly differ from the clock speed when most cores are used. For the multi-threaded benchmarks, we use different client counts to ensure that each architecture is fully utilized. As such, the resulting performance cannot be directly compared.

4.3 End-to-End Evaluation in Hyrise

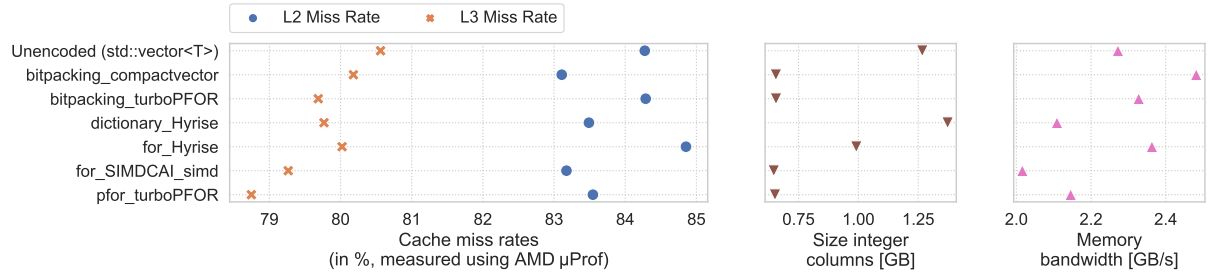
We compare the best-performing codecs from Section 4.1 and segment encodings already implemented in Hyrise. In Hyrise, Dictionary Encoding is the default encoding, with byte-aligned null suppression of the attribute vector (AV, cf. [16]). Dictionary Encoding is widely used in analytical databases such as SAP HANA [16], MonetDB [20], and HyPer [24]. We only encode the integer columns with the respective codec, and all other columns are dictionary-encoded. Then we measure the performance of Hyrise using the TPC-H, TPC-DS[4], and Join Order benchmarks[5]. The TPC-H and TPC-DS benchmarks are executed using a scale factor of 10. **??** displays the benchmark results. We discard results for LZ4 as it is mainly used for string columns in Hyrise. For most integer columns, LZ4 does not improve the compression ratio significantly and is up to 10× slower that other encodings. The performance drop is mainly caused by slow random accesses due to the block-based format of LZ4 (cf. [6]).

4.3.1 Memory Consumption. Compared to Dictionary Encoding, the lightweight integer encodings use up to 53 % less memory to store the integer columns. In the TPC-DS benchmark, the size reductions of the schemes are smaller, but even here, only *for_Hyrise* and *Unencoded* use more memory than Dictionary Encoding. As can be seen in Figure 8, the segments in the TPC-DS benchmark contain fewer distinct values on average compared to the TPC-H and JOB benchmarks. Fewer distinct values cause a smaller dictionary size and are more suitable for Dictionary Encoding. In general, the

[4] As of August 2021, Hyrise supports 41 of 99 TPC-DS queries.
[5] The source code that was used to implement the encodings in Hyrise is publicly available: https://github.com/benrobby/hyrise/



(a) Intel Xeon Gold 6240L



(b) AMD EPYC 7742

Figure 7: Cache access and bandwidth metrics for various integer segment encodings and TPC-H (multi-threaded, SF 10).

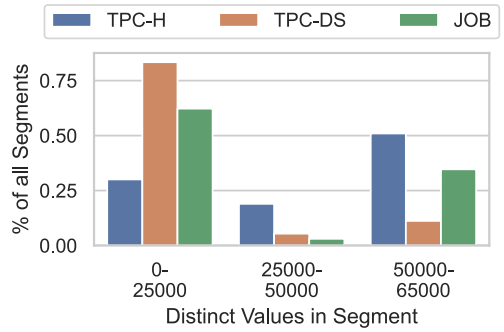


Figure 8: Distinct values per segment, showing TPC-DS has on average fewer distinct values.

most space-efficient implementations of *FoR*, *PFoR*, and *bitpacking* achieve similar compression rates in each of the three benchmarks.

4.3.2 Runtime Performance. In most benchmarks, *bitpacking_turboPFOR* and *bitpacking_compactvector* have a smaller total runtime than *pfor_turboPFOR* and *for_SIMDCAI_simd*. This finding is in line with the random access decoding results we observed earlier. As shown for TPC-H in Figure 5, almost all alternatives improve on performance as well as memory consumption compared to Dictionary Encoding. *Unencoded* (`std::vector<T>`) performs best in the single-threaded setting but is slower in the multi-threaded setting due to its comparatively large size and the bandwidth bottleneck. Compared to Dictionary Encoding, *bitpacking_compactvector* achieves a 5 - 16 % shorter runtime in the TPC-H benchmark. With the exception of the AMD CPU, it is the best performing scheme for multi-threaded settings.

For TPC-DS and Join Order Benchmark, the picture looks different as shown in Figure 6. For TPC-DS, Dictiontary Encoding is among the best schemes both for single- as well as multi-threaded settings. For the Join Order Benchmark, *bitpacking_compactvector* is the fastest scheme while also being among the smallest.

There is no single conclusion possible for the usage of Dictionary Encoding and the three benchmark workloads. While it does not perform well in TPC-H, it is among the best schemes for TPC-DS, and acceptable for the Join Order Benchmark.

In regards to the higher runtime of Dictionary Encoding, this may be influenced by the number of memory accesses it needs to perform. Dictionary Encoding performs two data-dependent lookups, which is why its performance deteriorates when the dictionary does not fit into the last level cache of the processor [31]. This problem is less pronounced in the other integer encodings. *FoR* and *PFoR* also need to perform two lookups, one for the offset value, and one for the reference value of the block. However, due to the small number of reference values, they can be kept in the cache more easily. *bitpacking* and *Unencoded* (`std::vector<T>`) only need to perform one lookup. In Figure 7, we see that in the TPC-H benchmark, Dictionary Encoding has indeed the most L2 and L3 cache misses, as we would expect. We also notice that the *bitpacking* schemes tend to have fewer cache misses than *FoR* and *pfor*, which also fits to the simpler access patterns of *bitpacking*.

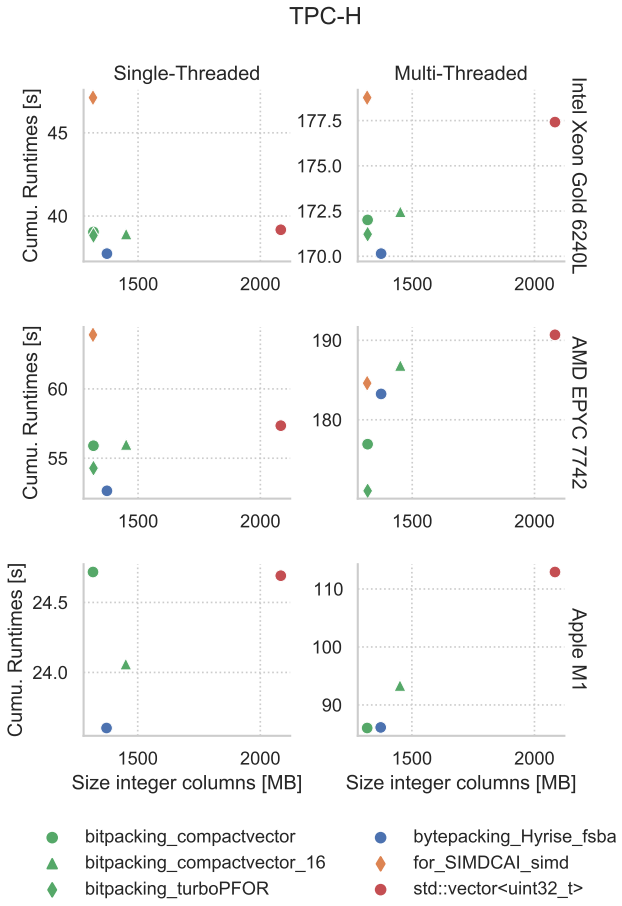


Figure 9: Size of integer columns and runtime performance of various vector compression schemes (TPC-H SF 10) for the three evaluated CPU architectures, single- and multi-threaded.

5 EVALUATION OF SECOND-ORDER VECTOR COMPRESSION TECHNIQUES AND IMPLEMENTATIONS IN HYRISE

In Hyrise, an abstraction for further auxiliary compression is used on top of the existing column encodings discussed in Section 4. We will refer to this abstraction as *vector compression*. Use cases of this second-level compression in Hyrise include Dictionary Encoding, where the sequence of dictionary indices, the Attribute Vector (AV), can be compressed. A further application is the compression of offset vectors for FoR and LZ4 column encodings. In the following, we will focus on the usage in Dictionary Encoding and evaluate how integer compression techniques affect DBMS performance.

5.1 Attribute Vector Characteristics and Access Patterns

The Attribute Vectors (AV) that we want to compress in Hyrise contain as many elements as the enclosing chunk. While the user can change the chunk size at runtime, it defaults to $C = 2^{16} - 1$. The integer values in the AV can reach from 0 to D, the dictionary size, which is smaller than C. Therefore, we can efficiently represent the AV indices using two bytes per value, assuming $D = 2^{16} - 1$.

When only a few distinct values are in a segment, the dictionary has significantly fewer elements than C. In these cases, a more compact representation of the values in the AV with fewer bits can be worth additional compression overhead. In the opposite case, with no duplicates in the segment, other encodings can be more efficient than Dictionary Encoding.

Depending on the size and duplication of the segment's values, the size of the AV makes up a significant part of the memory consumption. Hence, it makes sense to apply compression to it. It is also possible to compress the dictionary entries further, but this is not done in Hyrise.

Since we apply this vector compression on top of other encoding schemes, the need for a performant, lightweight compression is of even higher importance.

The access patterns for Dictionary AVs are the same as described in Section 4.3. We have to execute a lookup in the AV and a subsequent data-dependent lookup in the dictionary for each retrieval of a value. For this reason, we can apply the learnings from Section 4 and only explore the compression schemes that support random access while offering a good performance.

5.2 End-to-End Evaluation in Hyrise

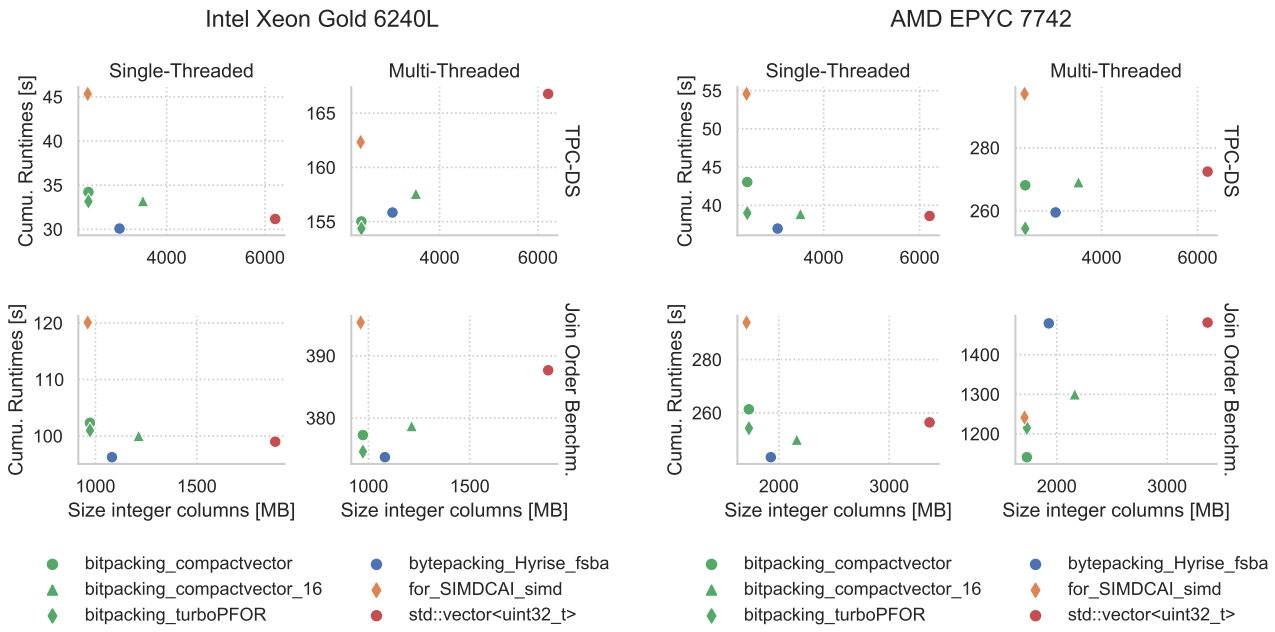
In the benchmarked scenario, we Dictionary-encode all columns to measure the effects of compressing the attribute vector (cf. Section 4.3).

5.2.1 Benchmark Results. For the TPC-H results shown in Figure 9, we see that the *bitpacking_compactvector* and *bitpacking_turboPFOR* libraries' runtime performance lies within +7 % to the existing baseline *bytepacking_Hyrise_fsba* for Intel and AMD. In contrast, for the Apple M1, both bitpacking approaches perform worse that Dictionary Encoding in the single-threaded setting. Furthermore, the uncompressed `std::vector<uint32_t>` performs well in the single-threaded setting on Intel and AMD, but performs much worse in the multi-threaded setting as it is limited by the CPU's bandwidth.

For most settings, *bytepacking_Hyrise_fsba* provides a good trade-off as it is among the smallest encodings and mostly performs well. The only exception is the the multi-threaded setting on the AMD, where both bitpacking approaches perform better.

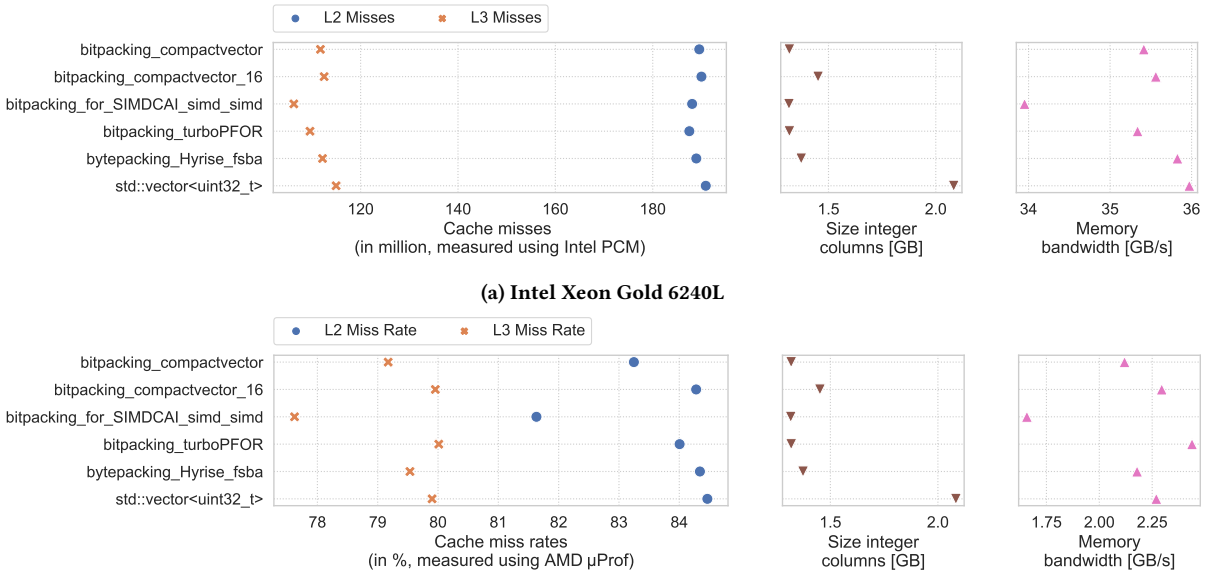
Similar to the integer segment encoding results, using `SIMDCAI` is not a good trade-off as alternatives are only slightly larger but perform significantly better. For each scheme, we also implemented a version where we decode the whole AV sequentially and then access the indices. These versions perform strictly worse than the corresponding random access implementation. Hence, we do not study them further.

For the total memory sizes in TPC-H, we observe that the *bitpacking_turboPFOR* and *for_SIMDCAI_simd* algorithms reduce the overall memory consumption compared to `std::vector<uint32_t>` by 27 %. Furthermore, *bitpacking* algorithms compress 4 % better than *bytepacking*. Depending on the number of distinct values, the ratio largely differs. Here, the primary key columns dominate the memory consumption, leaving less compression potential for bitpacking.



(a) Intel Xeon Gold 6240L **(b) AMD EPYC 7742**

Figure 10: Size of integer columns and runtime performance of various vector compression schemes (TPC-DS SF 10 and Join Order Benchmark), single- and multi-threaded.



(a) Intel Xeon Gold 6240L

(b) AMD EPYC 7742

Figure 11: Cache access and bandwidth metrics for various vector compression schemes and TPC-H (multi-threaded, SF 10).

5.2.2 Bitpacking and Bytepacking Memory Consumption. Upon detailed comparison of the bitpacking-compressed to the bytepacking-compressed segments in Figure 12, we observe significantly higher memory consumption reductions compared to *bytepacking_Hyrise_fsba* for chunks with non-byte-aligned distinct value count. We see reductions up to a factor of eight when only one bit is required to represent entries of the AV.

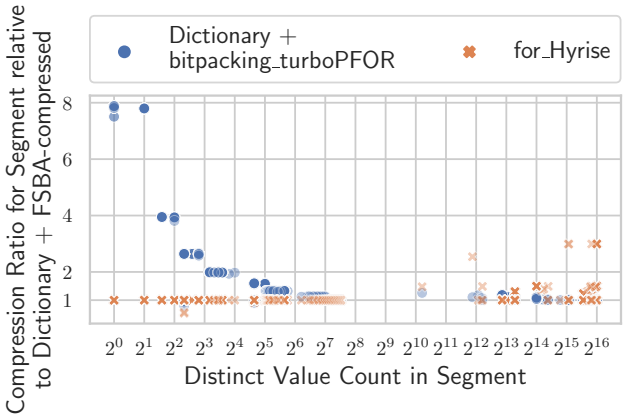


Figure 12: *bitpacking_turboPFOR* and *for_Hyrise* compression ratio relative to baseline *bytepacking_Hyrise_fsba*, per segment. Data for TPC-DS benchmark with Scale Factor 10.

Those advantages of bitpacking come into play very frequently. In fact, in the TPC-H dataset, in 57 % of all segments, the AV has $\geq 2^{10}$ times more entries than the dictionary while the distinct value counts are not byte-aligned. Interestingly, the data distributions where bitpacking excels are the same cases with high duplication where Dictionary Encoding already outperforms other encodings such as FoR. Hence, we expect to see even more significant memory size reductions with compression configurations that chose better-suited encodings for segments where Dictionary Encoding performs poorly (i.e., primary key columns).

5.2.3 Performance Analysis. The end-to-end runtime varies significantly for different compression algorithms. We explain this variance with two factors: CPU load and memory access. Several algorithms perform better than `std::vector<uint32_t>` although they add decoding overhead. Our analysis of L2 and L3 cache loads and misses in Figure 11 shows that these metrics mostly correlate with the compression schemes' memory consumption.

Still, improved caching behavior does not explain all runtime differences between the algorithms. Especially when we regard *for_SIMDCAI_simd*, we see that it has a low memory consumption and few cache accesses/misses but still performs considerably worse than *bitpacking_turboPFOR*, with a similar compression rate. We explain this difference with the codecs' higher CPU decoding load as shown in Section 4.1, which results in a lower memory bandwidth usage since the CPU can not issue enough reads.

6 SELECTION OF ENCODING AND VECTOR COMPRESSION SCHEMES

The results shown Sections 4 and 5 emphasize that the decision which segment encoding or vector compression to use is not trivial. Each approach on the pareto frontier is a viable alternative when runtime performance and memory consumption need to be balanced, as no other alternative dominated in runtime performance *and* size. The effects of multi-threading on cache miss rates and the DRAM bandwidth further complicate the selection. Moreover, the analyzed approaches sometimes differ significantly in their runtime performance on different platforms.

Several recent publications have recognized this issue and propose automated selection methods. These methods optimize runtime performance and memory consumption by analyzing data and access patterns in order to find the best configuration of a database instance [4, 8, 11, 21]. Most approaches either use database-internal access counters (e.g., Hyrise or SAP HANA [7]) or analyze the database's query plan cache [4]. To estimate the runtime performance, several approaches use calibration phases and machine learning to create prediction models that allow estimating the performance for a given system [4, 28]. While simple heuristics (cf. decision tree of Abadi et al. [1]) are helpful (e.g., to avoid dictionary encoding for columns with many distinct values), more sophisticated methods are required to ensure robust runtime performance or allow optimizing compression schemes with certain constraints (e.g., on memory budget or runtime).

In the long run, automated compression selection needs to be combined with other methods for footprint reduction, such as data tiering. If and how well various automated optimization methods can be combined is studied in various previous and current projects (cf. [22, 37]).

7 CONCLUSION

We have analyzed integer compression schemes in a synthetic scenario as well as in various end-to-end database benchmarks. While most libraries perform well for sequential accesses, e.g., by vectorization using SIMD instructions, accesses are often different database systems. We have shown that many accesses in the columnar in-memory database Hyrise are performed as random accesses when executing TPC-H, TPC-DS, and JOB. For these workloads, integer compression techniques with fast random access performance, such as simple *bitpacking*, often outperformed widely used schemes such as Dictionary Encoding. For columns other than integer columns, we can compress Dictionary Encoding's attribute vector further with second-order vector compression techniques such as bit-aligned null suppression. This technique is especially effective in the canonical use case of Dictionary Encoding with only a few distinct values in a segment where it reduces the memory usage significantly compared to byte-aligned compression techniques. The results emphasize that cost-efficient database systems need to be aware of different encoding alternatives to find the best balance between runtime performance and space consumption.

ACKNOWLEDGMENTS

We thank Tobias Pape and the HPI Data Engineering Lab for supporting us with the cache miss analyses and providing access to the evaluated hardware platforms.

REFERENCES

[1] Daniel J. Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Chicago, Illinois, USA, June 27-29, 2006*. 671–682. https://doi.org/10.1145/1142473.1142548

[2] Daniel J. Abadi, Daniel S. Myers, David J. DeWitt, and Samuel Madden. 2007. Materialization Strategies in a Column-Oriented DBMS. In *Proceedings of the 23rd International Conference on Data Engineering, ICDE 2007, The Marmara Hotel,*

Istanbul, Turkey, April 15-20, 2007. 466–475. https://doi.org/10.1109/ICDE.2007.367892

[3] Haoqiong Bian, Ying Yan, Wenbo Tao, Liang Jeff Chen, Yueguo Chen, Xiaoyong Du, and Thomas Moscibroda. 2017. Wide Table Layout Optimization based on Column Ordering and Duplication. In *Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017*. 299–314. https://doi.org/10.1145/3035918.3035930

[4] Martin Boissier and Max Jendruk. 2019. Workload-Driven and Robust Selection of Compression Schemes for Column Stores. In *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*. 674–677. https://doi.org/10.5441/002/edbt.2019.84

[5] Martin Boissier, Rainer Schlosser, and Matthias Uflacker. 2018. Hybrid Data Layouts for Tiered HTAP Databases with Pareto-Optimal Data Placements. In *34th IEEE International Conference on Data Engineering, ICDE 2018, Paris, France, April 16-19, 2018*. 209–220. https://doi.org/10.1109/ICDE.2018.00028

[6] Peter A. Boncz, Thomas Neumann, and Viktor Leis. 2020. FSST: Fast Random Access String Compression. *Proc. VLDB Endow.* 13, 11 (2020), 2649–2661. http://www.vldb.org/pvldb/vol13/p2649-boncz.pdf

[7] Michael Brendle, Nick Weber, Mahammad Valiyev, Norman May, Robert Schulze, Alexander Böhm, Guido Moerkotte, and Michael Grossniklaus. 2021. Precise, Compact, and Fast Data Access Counters for Automated Physical Database Design. In *Datenbanksysteme für Business, Technologie und Web (BTW 2021), 19. Fachtagung des GI-Fachbereichs „Datenbanken und Informationssysteme" (DBIS), 13.-17. September 2021, Dresden, Germany, Proceedings*. 79–100. https://doi.org/10.18420/btw2021-04

[8] Lujing Cen, Andreas Kipf, Ryan Marcus, and Tim Kraska. 2021. LEA: A Learned Encoding Advisor for Column Stores. In *aiDM '21: Fourth Workshop in Exploiting AI Techniques for Data Management, Virtual Event, China, 25 June, 2021*. 32–35. https://doi.org/10.1145/3464509.3464885

[9] Yann Collet. 2021. LZ4 - Extremely fast compression. https://lz4.github.io/lz4/ [Online; accessed 2021-08-09].

[10] Patrick Damme, Dirk Habich, Juliana Hildebrandt, and Wolfgang Lehner. 2017. Lightweight Data Compression Algorithms: An Experimental Survey (Experiments and Analyses). In *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017*. 72–83. https://doi.org/10.5441/002/edbt.2017.08

[11] Patrick Damme, Annett Ungethüm, Juliana Hildebrandt, Dirk Habich, and Wolfgang Lehner. 2019. From a Comprehensive Experimental Survey to a Cost-based Selection Strategy for Lightweight Integer Compression Algorithms. *ACM Trans. Database Syst.* 44, 3 (2019), 9:1–9:46. https://doi.org/10.1145/3323991

[12] Patrick Damme, Annett Ungethüm, Johannes Pietrzyk, Alexander Krause, Dirk Habich, and Wolfgang Lehner. 2020. MorphStore: Analytical Query Engine with a Holistic Compression-Enabled Processing Model. *Proc. VLDB Endow.* 13, 11 (2020), 2396–2410. http://www.vldb.org/pvldb/vol13/p2396-damme.pdf

[13] Jeffrey Dean. 2009. Challenges in building large-scale information retrieval systems: invited talk. In *Proceedings of the Second International Conference on Web Search and Web Data Mining, WSDM 2009, Barcelona, Spain, February 9-11, 2009*. 1. https://doi.org/10.1145/1498759.1498761

[14] Markus Dreseler, Martin Boissier, Tilmann Rabl, and Matthias Uflacker. 2020. Quantifying TPC-H Choke Points and Their Optimizations. *Proc. VLDB Endow.* 13, 8 (2020), 1206–1220. https://doi.org/10.14778/3389133.3389138

[15] Markus Dreseler, Jan Kossmann, Martin Boissier, Stefan Klauck, Matthias Uflacker, and Hasso Plattner. 2019. Hyrise Re-engineered: An Extensible Database System for Research in Relational In-Memory Data Management. In *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019*. 313–324. https://doi.org/10.5441/002/edbt.2019.28

[16] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. 2012. The SAP HANA Database – An Architecture Overview. *IEEE Data Eng. Bull.* 35, 1 (2012), 28–33. http://sites.computer.org/debull/A12mar/hana.pdf

[17] Franz Färber, Alfons Kemper, Per Åke Larson, Justin Levandoski, Thomas Neumann, and Andrew Pavlo. 2017. Main Memory Database Systems. *Foundations and Trends in Databases* 8, 1-2 (2017), 1–130. https://doi.org/10.1561/1900000058

[18] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. 1998. Compressing Relations and Indexes. In *Proc. ICDE*. 370–379.

[19] IDG. 2020. 2020 Cloud Computing Study. https://www.idg.com/tools-for-marketers/2020-cloud-computing-study/. [Online; accessed 2021-08-09].

[20] Stratos Idreos et al. 2012. MonetDB: Two Decades of Research in Column-oriented Database Architectures. *IEEE Data Eng. Bull.* 35, 1 (2012), 40–45.

[21] Hao Jiang, Chunwei Liu, John Paparrizos, Andrew A. Chien, Jihong Ma, and Aaron J. Elmore. 2021. Good to the Last Bit: Data-Driven Encoding with CodecDB. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. 843–856. https://doi.org/10.1145/3448016.3457283

[22] Jan Kossmann and Rainer Schlosser. 2020. Self-driving database systems: a conceptual approach. *Distributed Parallel Databases* 38, 4 (2020), 795–817. https://doi.org/10.1007/s10619-020-07288-w

[23] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. 2012. The Vertica Analytic Database: C-Store 7 Years Later. *Proc. VLDB Endow.* 5, 12 (2012), 1790–1801. https://doi.org/10.14778/2367502.2367518

[24] Harald Lang, Tobias Mühlbauer, Florian Funke, Peter A. Boncz, Thomas Neumann, and Alfons Kemper. 2016. Data Blocks: Hybrid OLTP and OLAP on Compressed Storage using both Vectorization and Compilation. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*. 311–326. https://doi.org/10.1145/2882903.2882925

[25] D. Lemire and L. Boytsov. 2013. Decoding billions of integers per second through vectorization. *Software: Practice and Experience* 45, 1 (May 2013), 1–29.

[26] Daniel Lemire, Leonid Boytsov, and Nathan Kurz. 2015. SIMD compression and the intersection of sorted integers. *Software: Practice and Experience* 46, 6 (Apr 2015), 723–749.

[27] Daniel Lemire, Nathan Kurz, and Christoph Rupp. 2018. Stream VByte: Faster byte-oriented integer compression. *Inform. Process. Lett.* 130 (Feb 2018), 1–6.

[28] Lin Ma, William Zhang, Jie Jiao, Wuwen Wang, Matthew Butrovich, Wan Shen Lim, Prashanth Menon, and Andrew Pavlo. 2021. MB2: Decomposed Behavior Modeling for Self-Driving Database Management Systems. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*. 1248–1261. https://doi.org/10.1145/3448016.3457276

[29] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. 2000. Optimizing database architecture for the new bottleneck: memory access. *The VLDB Journal* 9, 3 (2000), 231–246.

[30] Microsoft. 2021. SLA for Azure Cosmos DB. https://azure.microsoft.com/en-us/support/legal/sla/cosmos-db/ [Online; accessed 2021-08-09].

[31] Georgios Psaropoulos, Thomas Legler, Norman May, and Anastasia Ailamaki. 2019. Interleaving with coroutines: a systematic and practical approach to hide memory latency in index joins. *VLDB J.* 28, 4 (2019), 451–471. https://doi.org/10.1007/s00778-018-0533-6

[32] Keven Richly, Rainer Schlosser, and Martin Boissier. 2021. Joint Index, Sorting, and Compression Optimization for Memory-Efficient Spatio-Temporal Data Management. In *37th IEEE International Conference on Data Engineering, ICDE 2021, Chania, Greece, April 19-22, 2021*. 1901–1906. https://doi.org/10.1109/ICDE51399.2021.00174

[33] Mark A. Roth and Scott J. Van Horn. 1993. Database Compression. *SIGMOD Rec.* 22, 3 (Sept. 1993), 31–39. https://doi.org/10.1145/163090.163096

[34] Reza Sherkat, Colin Florendo, Mihnea Andrei, Rolando Blanco, Adrian Dragusanu, Amit Pathak, Pushkar Khadilkar, Neeraj Kulkarni, Christian Lemke, Sebastian Seifert, Sarika Iyer, Sasikanth Gottapu, Robert Schulze, Chaitanya Gottipati, Nirvik Basak, Yanhong Wang, Vivek Kandiyanallur, Santosh Pendap, Dheren Gala, Rajesh Almeida, and Prasanta Ghosh. 2019. Native Store Extension for SAP HANA. *Proc. VLDB Endow.* 12, 12 (2019), 2047–2058. https://doi.org/10.14778/3352063.3352123

[35] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Mühlbauer, Thomas Neumann, and Manuel Then. 2018. Get Real: How Benchmarks Fail to Represent the Real World. In *Proceedings of the 7th International Workshop on Testing Database Systems, DBTest@SIGMOD 2018, Houston, TX, USA, June 15, 2018*. 1:1–1:6. https://doi.org/10.1145/3209950.3209952

[36] Wayne Xin Zhao, Xudong Zhang, Daniel Lemire, Dongdong Shan, Jian-Yun Nie, Hongfei Yan, and Ji-Rong Wen. 2015. A General SIMD-Based Approach to Accelerating Compression Algorithms. *ACM Trans. Inf. Syst.* 33, 3 (2015), 15:1–15:28. https://doi.org/10.1145/2735629

[37] Daniel C. Zilio, Jun Rao, Sam Lightstone, Guy M. Lohman, Adam J. Storm, Christian Garcia-Arellano, and Scott Fadden. 2004. DB2 Design Advisor: Integrated Automatic Physical Database Design. In *(e)Proceedings of the Thirtieth International Conference on Very Large Data Bases, VLDB 2004, Toronto, Canada, August 31 - September 3 2004*. 1087–1097. https://doi.org/10.1016/B978-012088469-8.50095-4

[38] Jacob Ziv and Abraham Lempel. 1977. A Universal Algorithm for Sequential Data Compression. *IEEE Transactions on Information Theory* 23, 3 (May 1977), 337–343.