An Adaptive Column Compression Family for Self-Driving Databases

Marcell Fehér
Agile Cloud Lab, Department of Electrical and Computer Engineering, DIGIT, Aarhus University
Aarhus, Denmark
sw0rdf1sh@ece.au.dk

Daniel E. Lucani
Agile Cloud Lab, Department of Electrical and Computer Engineering, DIGIT, Aarhus University
Aarhus, Denmark
daniel.lucani@ece.au.dk

Ioannis Chatzigeorgiou
School of Computing and Communications,
Lancaster University
Lancaster, United Kingdom
i.chatzigeorgiou@lancaster.ac.uk

ABSTRACT

Modern in-memory databases are typically used for high-performance workloads, therefore they have to be optimized for small memory footprint and high query speed at the same time. Data compression has the potential to reduce memory requirements but often reduces query speed too. In this paper we propose a novel, adaptive compressor that offers a new trade-off point of these dimensions, achieving better compression than LZ4 while reaching query speeds close to the fastest existing segment encoders. We evaluate our compressor both with synthetic data in isolation and on the TPC-H and Join Order Benchmarks, integrated into a modern relational column store, Hyrise.

1 INTRODUCTION

Database systems employ a wide variety of compression schemes to reduce memory footprint, increase effective storage capacity and overcome bandwidth limitations of slow hard drives. With the proliferation of in-memory databases, the role of data compression has changed from a nice-to-have feature to an essential tool that is required to store and analyze large datasets. Since the maximum amount of memory is an order of magnitude smaller than hard drive capacity in most desktop and server computers, simply adding more memory cannot solve the problem anymore.

Additionally, more and more workloads move to the cloud where pricing models are dictated by the cloud provider. Hardware resources available for cloud-based virtual machines are typically pre-configured sets, where the price is proportional to the amount of memory and CPU included in each machine type. Therefore, efficient compression directly affects operational costs of in-memory databases running in cloud environments.

Compression methods decrease data size by exploiting redundancy present within the input data, therefore column-oriented databases are more suitable to compression than traditional row stores. This is because a list of values from the same column always have the same data type and more likely to be compressible than records with multiple fields of different types. Modern in-memory column stores take this a step further and split each column to segments, where each segment has its own encoding and managed separately from the others. This approach enables great performance for both transactional and analytical workloads by using a write-optimized encoding of the most recent segment (where new data is added) and a read-optimized one for older segments.

A commonly used segment encoding strategy is assigning a single encoder per data type (integer, string, etc.) and use it for all segments in the database. While this decreases memory consumption and in some cases speeds up queries, it does not take into consideration the characteristics of the stored data. There is some research discussing dynamic encoding selection. CodecDB [13] trains a model that infers which compressor is most likely to achieve the highest compression based on static metrics derived from the target column, such as cardinality or domain. Cen *et al.* proposed a system called Learned Encoding Advisor [5], which takes into consideration the values (both statistics and a 1% extract), sample queries and the underlying hardware as well, when selecting the best encoder. It is evaluated with two high-level strategy options: maximum compression and highest query performance. Boissier designed an encoding selection module for Hyrise, which learns cost models of encoders from the physical query cache, and provides the best segment encoder for a given memory budget [4].

In this paper we introduce a novel segment encoder family for integer columns which determines its own best parameters based on the data to be encoded. We evaluate them against the most commonly used segment compressors, both in isolation using synthetic data, and in a fully featured relational column store on the TPC-H [2] and Join Order Benchmarks [14], analyzing the effects of different encoding parameters. Extending the idea of finding the best parameter for a single encoder to the whole database, we propose a segment encoding scheme for autonomous databases where the compressor selection is driven by query patterns to maximize performance.

The paper is organized as follows. In section 2 we review the current state of segment encoding in modern relational databases and introduce the data compression technique that our encoders are based on. Then, in section 3 we present the detailed design of four segment variants using this compression algorithm. Our proposal for an intelligent, adaptive segment encoding framework for autonomous databases is presented in section 4. The results of evaluating the new segment types against the commonly used ones, both in isolation and with industry-standard analytical benchmarks are shown in section 5. Finally, section 6 draws conclusions from the results and proposes future work.

2 BACKGROUND

2.1 Segment Encoding in In-Memory Databases

As companies collect and analyze increasingly large data sets, the performance of disk-based databases is not satisfactory anymore for many of them. With the decreasing price and increasing size and speed of main memory, in-memory databases have become viable and affordable alternatives [7] of disk-based systems. They are ideal for performance-critical workloads, which means processing speed and memory footprint are the new important metrics [17]. Compression has the potential to markedly decrease the footprint, especially in modern columnar databases like SAP HANA [8], HyPer [10] or DuckDB [20], which horizontally partition data into segments and the unit of encoding is a segment instead of the whole column. Therefore, compressors enjoy two extra benefits in this environment: they can work on an array of values of the same type, and data distribution within a segment is more likely to be self-similar than in the whole column. Both of these qualities work in favor of data compressors. However, as decompression requires extra processing, it typically comes at a cost of query performance. Therefore, it is critical to select the most appropriate compressor for columns that are heavily used in queries. Since query performance in analytical workloads mostly depends on processing speed of integer columns [12], we focus on lossless integer compression.

Note, that strings make up the majority of data in both real world datasets and widely used OLAP benchmarks like TPC-H and TPC-DS. In many cases people store numeric columns like timestamps, booleans or floats as strings as well, and cast them in the SQL query [22]. Therefore, integer compression usually has a minor effect on the overall storage footprint. However, there are several major use cases where numeric data is the dominant type, for example IoT time series. Consequently, our goal is to reduce data size and retain or increase speed of frequent operations on integer columns at the same time.

We assess our proposed segment encoders against the most commonly used integer compression schemes in columnar databases.

- Dictionary Encoding [3] is a widely used data compression technique where unique values are collected to a sorted dictionary, and their occurrences in the input data vector are represented by a list of dictionary offsets. It is the default encoding scheme of several column stores.
- Frame-of-Reference (FoR) [11] encoding stores difference of each value to the common minimum, as well as the minimum itself. The delta values are bit packed to the smallest possible width. For evaluations we used an improved version is this technique, called Patched FoR (or PFoR). It splits the input data to fixed size blocks and performs FoR on each one separately, hoping to exploit local similarities present in the (large) array.
- LZ4 [1] is a relatively heavy statistical compressor that offers high degree of data size reduction for slower compression and decompression than the other two lightweight methods. This scheme is typically used to compress columns that never or very rarely participate in query conditions.

To test our segment compressor in a real system, we use Hyrise [6], a modern relational in-memory column store. It is an open-source research database with an extensible framework for segment encoders. All three encodings above are standard built-in options, making it easy for a segment developer to run benchmarks against them. Even though vectorized implementations of some of these algorithms are available via libraries [23], we are using the non-SIMD variants for our comparisons.

2.2 Generalized Deduplication

Dictionary encoding, paired with null suppression of the dictionary, attribute vector or both, achieves high compression when the cardinality of the input data is sufficiently low (e.g., there are few unique values). However, when applied to datasets where all values are different, like a primary key column in a relational database, it inflates the data and slows down every operation. Several methods have been developed for compressing high cardinality datasets, each with their own expectations about the value distribution. For example, FoR and PFoR encodings assume a narrow domain of encoded values.

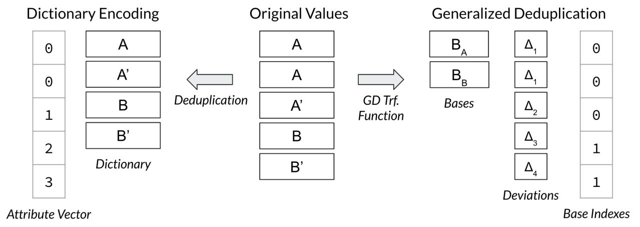


Figure 1: Dictionary Encoding and Generalized Deduplication of the same set of values. GD uses an arbitrary, user-specified transformation function to convert input values to bases and deviations.

A new technique, introduced by Vestergaard *et al.* [21] and referred to as *Generalized Deduplication* (GD) does not have any inherent assumptions about the data distribution. It prodives a flexible compression framework that can be tailored for the data at hand. Compression with GD is a two-step process: first, every input value is passed through a user-defined, arbitrary *transformation function* that converts it to a pair of *base* and *deviation*, where similar inputs should generate the same base with different deviations. The goal of this function is to separate the identical and varying parts of the input data chunks. Think of it as a booster step in dictionary encoding, which aims to increase the deduplication rate. Secondly, bases are deduplicated and the base index of each original value is determined (see Figure 1). Decompression is a straightforward process in the opposite direction: the base and deviation is looked up and passed to the inverse transformation function, which reconstructs the original value. Generalized Deduplication has the following key properties:

- It can be either a lossless or a lossy compressor, depending on whether the deviations are kept or discarded.
- Dictionary Encoding is a special case of GD, where the transformation is the identity function.
- GD supports constant time random access, since reconstructing a single value given its index requires 3 lookups: the

base index, the deviation and the base itself (using the base index).
- Arbitrary assumptions about the input data characteristics can be encoded as the transformation function.
- Null suppression (removing leading zeroes, see Figure 2) can be applied to any combination of the bases, deviations and base indexes.
- The customizability of GD makes it a good fit in a variety of use cases, including file systems, cloud storage, time series or relational databases. It is applicable in every scenario when the similarity between input data chunks can be extracted with a reversible transformation.

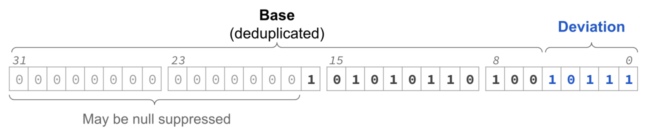


Figure 2: Decomposition of a 4-byte unsigned integer (87703) to base (2740) and deviation (23) using the LastBit transformation with 5 bit deviations.

GD-based compressors have been proposed in the past for different data and use cases (e.g., [9, 18]), but not as a segment encoder in relational databases. For integer compression in this environment, we chose a simple but powerful transformation function that considers the n least significant bits of a data value as deviation, and the previous bits as base (see Figure 2). When applied to a series of values, this transformation assumes they are close to each other, e.g. the bits with the highest variation across the whole segment are at the end of the values. This transformation function, called *LastBit*, partitions the range of integers to consecutive regions of size 2^n, that is, $[i2^n \ldots (i+1)2^n - 1]$ for $i = 0, 1, \ldots$, where the base of every value within region i is $i2^n$ (the smallest element of the region). For example with 6 bit deviations each base covers 64 consecutive values (see Figure 3). It has an additional advantage besides simplicity (therefore, processing speed): it is order preserving for both bases and deviations, e.g., if $x > y$ then either $Base(x) > Base(y)$ or $Base(x) = Base(y)$ and $Deviation(x) > Deviation(y)$. This property gives LastBit an advantage over other transformation functions in relational databases, since it enables faster predicate evaluations in certain cases.

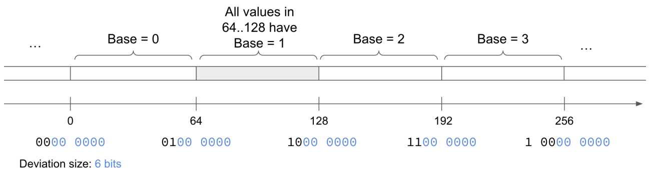


Figure 3: Partitioning integers with LastBit transformation using 6-bit deviations.

3 GD SEGMENTS

We propose four designs based on Generalized Deduplication with the LastBit transformation function to store, and access and query numeric segments in a column store (Figure 4). When a segment is encoded, each value is split into a base and a deviation. The deviation size is an input parameter of the segment constructor, given in bits. All segment variants include the deduplicated and sorted list of bases, but they store deviations and base indexes in different ways. The segments are implemented in C++17, where we used `compact::vector`[1] for bit-packed lists and `std::vector` for regular ones. Bases and deviations are bit-packed lists in all four variants, using exactly [32-deviation size] bits per base and [deviation size] bits per deviation.

GD Segment 1 has the simplest internal structure where each deviation is stored separately. Both deviations and base indexes contain one entry per segment value, in the same order as the original data. For example, to reconstruct the first value of the segment, we need to combine the base indicated by the first base index with the first deviation. Base indexes is a bit-packed list just like the bases and deviations, but its width is determined in run-time to fit the largest index.

In **GD Segment 2** the deviations are also deduplicated and stored in a sorted list. Since we cannot find the deviation of value i at $deviations[i]$ any more, we must store the deviation index for each value offset explicitly. To minimize the number of memory lookups during a reconstruction, we store each base index - deviation index pair in a single value, and construct a bit-packed list of them. The width of both indexes (and therefore the width of the reconstruction list) is calculated in run-time, based on the number of unique bases and deviations present in the segment.

GD Segment 3 aims to reduce the number of bits needed to address deviations. Since in GD Segment 2 we stored all deviations in a single list, the deviation index in the reconstruction list must be wide enough for the largest one. Instead of a single list with all (unique) deviations, we can organize them for each base separately. We simply scan all base-deviation pairs produced by the segment values, and group the deviations by their bases. Finally, we remove the duplicates and sort. The resulting 2-dimensional array consists of the unique deviations for each base, which are shorter than the single global list of deviations, requiring fewer bits to address. Given that each base is associated with a local group of deviations, some deviations are likely to be members of multiple local groups and be stored multiple times.

GD Segment 4 has a drastically different structure that is geared towards fast table scans but is no longer random accessible. Similarly to GD Segment 3, deviations are collected per base and sorted, but in this case not deduplicated. For each deviation we also store the original value offset, referred to as chunk offset on Figure 4, which is needed to produce the result for table scans.

The design choices of the GD Segment variants are motivated by the characteristics of generalized deduplication, the LastBit transformation and our goal to optimize for compression, random access and table scan speed at the same time. The four presented designs stand at different trade-off points between these dimensions.

[1]https://github.com/gmarcais/compact_vector

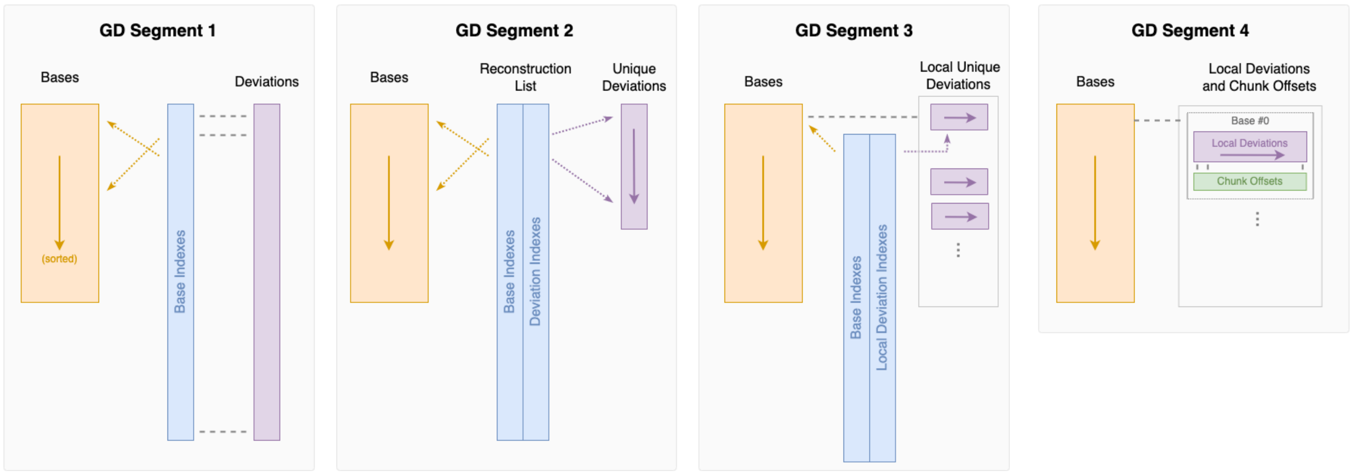


Figure 4: Proposed GD-based segment encoders.

3.1 Fast Table Scans

Query performance of a segment compressor primarily depends on the speed of two operations: random access (reconstruct the original value given its offset, also known as *dereferencing*) and table scan (given a predicate and a query value, return the segment offsets that satisfy the condition). In Table 1 we have provided the formulas and number of memory lookups per random access, which directly affects random access speeds. Scans can be performed without any special support from the segment encoder, simply by iterating or decompressing the whole segment first, and evaluating the predicate on each value, one by one. While this yields correct results and is a reasonable default behavior, it can be slower than exploiting the internal structure of the encoded representation, if possible. As we already hinted, all GD Segment versions offer custom table scan implementations that do not require decompressing raw values for predicate evaluation. In fact, not a single value needs to be reconstructed to serve scans with any predicate in either of the GD Segment variants.

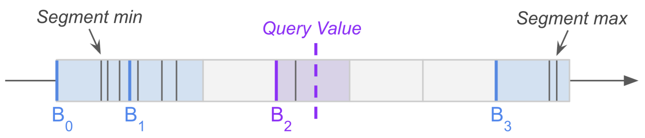


Figure 5: A sample segment with only 9 stored values and 4 bases

The key to evaluating predicates[2] without decompressing a GD Segment is the LastBit transformation function. Since it partitions integers to disjunct regions, we can quickly determine which base ranges may contain values that satisfy the current query. Figure 5 depicts a small segment containing only 9 values that map to 4 bases in total. When a table scan is performed, first the query value is passed through the same GD transformation function as the segment values did earlier, determining the query base and deviation. Next, searching for the query base in the bases list using `std::lower_bound` (or `std::upper_bound` depending on the predicate) tells us whether it is present in the segment, its index if present, or the index of the first larger base. Since the list of bases is sorted, we gain all this valuable information for a cheap, $O(logn)$ binary search. If the query base is present in the segment, this is the only base range that requires further investigation to evaluate the predicate. Values of all other base ranges are either completely contained in the result set, or not at all. For example, if the query predicate on Figure 5 is *GreaterEquals*, surely no value from base ranges B_0 and B_1 is a match, but every value of B_3 is. Whether a non-query value base range is included or excluded from the result set depends on the predicate. This zero-cost elimination can be done with all predicates, as it is a consequence of the order preserving nature of the LastBit transformation. If the query base is present in the segment, it needs to be scanned to find the values satisfying the predicate. Since deviations are also order preserving, there is no need to reconstruct the values of the query base. Comparing the stored deviations with the query deviation using the query predicate yields the correct results for the table scan.

Determining which deviations belong to the query base and comparing them to the query deviation is where the four GD Segment designs differ the most. GD Segment 1 is the slowest, since it does not store the deviations per base, therefore the whole base indexes list must be traversed to find the deviation indexes that map to the query base. In GD Segment 2 the process is identical, but as only unique deviations are stored, there is a minor performance gain due to the better cache hit rate. GD Segment 3 is faster, because stored deviations have been grouped by bases, therefore it does not have to iterate through the whole reconstruction list when scanning the deviations of the query base. Additionally, since local unique deviations are sorted, finding which ones satisfy the query condition requires a fast binary search. GD Segment 4 is able to evaluate table scans with at most two binary searches due to its data layout, granting it exceptional speed. Unfortunately, without a reconstruction list, it also lost the ability of constant-time random

[2]The following predicates are valid for integer columns: Equals, NotEquals, Greater, GreaterEquals, Less, LessEquals

Segment	**Formula to reconstruct value at of offset** *i (the GD inverse transformation is denoted by* $\bigoplus$*)*	Lookups
Uncompressed	`data[i]`	1
Dictionary	`dictionary[attribute_vector[i]]`	2
GD Segment 1	`bases[base_indexes[i]]` $\bigoplus$ `deviations[i]`	3
GD Segment 2	`bases[base_indexes[i]]` $\bigoplus$ `unique_devs[deviation_indexes[i]]`	3
GD Segment 3	`bases[base_indexes[i]]` $\bigoplus$ `local_unique_devs[base_indexes[i]] [local_dev_indexes[i]]`	4
GD Segment 4	*not random accessible*	-

Table 1: Random access formula and number of memory accesses for different segment types.

access, since there is no way to look up the base and deviation based on the value offset. Instead, it has to iterate over all offset lists until the requested one is found.

4 ADAPTIVE SEGMENT ENCODING

Generalized deduplication is unique among the commonly used encoding schemes in the sense that there is no widely applicable default configuration that yields predictable performance independent of the data. For GD Segments, the deviation size that results in good compression and query performance heavily depends on the data distribution. A database administrator could pick an arbitrary size as a global default (e.g., 8-bit deviations), but it likely won't be the best setting across different columns, segments of the same column, or even the same segment over its whole lifetime. Thus, a fixed default almost certainly wastes memory and CPU cycles eventually.

Instead of relying on an administrator to manually select the deviation size (either globally, or per-segment), we propose an iterative encoder for GD Segments. It automatically determines the best deviation size by trying all values from 1 to 30 bits (assuming 32-bit integer data) and selects the best one for the segment. The only problem is how to define the "best". It seems trivial to use the deviation size that achieves the highest compression, however, the best query performance is not necessarily at the same setting. This is due the different internal structures and custom table scan logic of GD Segment variants. The best option can only be determined based on testing the performance of different deviation sizes and knowing the relative importance of multiple factors: *compression*, as well as the speed of *random access*, *sequential access* and *table scan*. Even compression and decompression speeds are relevant in databases where segment encoding (and re-encoding) is a blocking operation.

Our iterative GD Segment encoder works by first encoding the input data with all 30 possible deviation sizes. It records the compression and runs a series of speed tests to determine the performance of each option. We measure sequential access (by dereferencing all offsets), random access (by requesting a set of random offsets), and table scans using the six integer predicates and random query values. The result of diagnostic tests is a table similar to Table 2.

Based on these measurements, there are multiple ways to select the best deviation size. If the relative importance of the dimensions is available, we can consider them as weights and find the deviation size that yields the best combination of compression and speeds. Figure 6 shows a few examples of weight distributions. These can be fixed based on the database architecture, or even better, inferred from the workload experienced by the segment while it was unencoded.

Databases that use *late materialization* (like Hyrise) execute queries in a way that lists of segment offsets are passed between processing nodes (e.g., a table scan or a join) and actual values are materialized as late as possible. As a result, random and sequential access via iterator dereferencing are the dominant segment operations. An alternative execution strategy is to materialize values at the very first operation (typically a table scan) and and pass them directly during evaluation. Therefore, *early materialization* databases achieve better query performance if they select an encoding with more weight on table scan speed and less on random access.

Relational databases have long been collecting statistics to guide query planning. We propose extending this capability with high granularity access and table scan statistics on a per-segment basis. Specifically, recording the number of sequential and random accesses, as well as table scans with each predicate separately. If this detailed query history is available, the system can use it in the beginning of the segment encoding process to infer the relative importance of performance metrics, which provides the weights for selecting the most appropriate compressor configuration.

Moreover, performance-based encoding selection can be extended to all segment encoders and the complete lifetime of segments. A *self-driving database* [15, 16, 19] is a relatively new concept

Dev.Size	Comp.	Seq.Access	Rand.Access	TableScan
1 bit	2%	8 ns	17 ns	171 μs
2 bits	27%	8 ns	16 ns	160 μs
3 bits	39%	10 ns	121 ns	159 μs
		...		
30 bits	3%	9 ns	16 ns	259 μs

Table 2: Sample results of GD Segment 1 diagnostic tests on a primary key segment

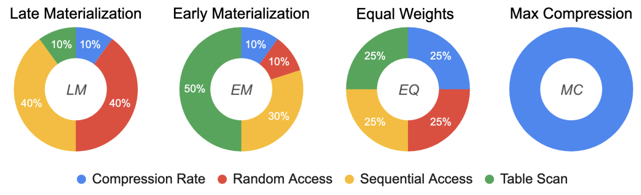


Figure 6: Possible presets of relative importance between compression and query performance, when selecting the best segment encoder.

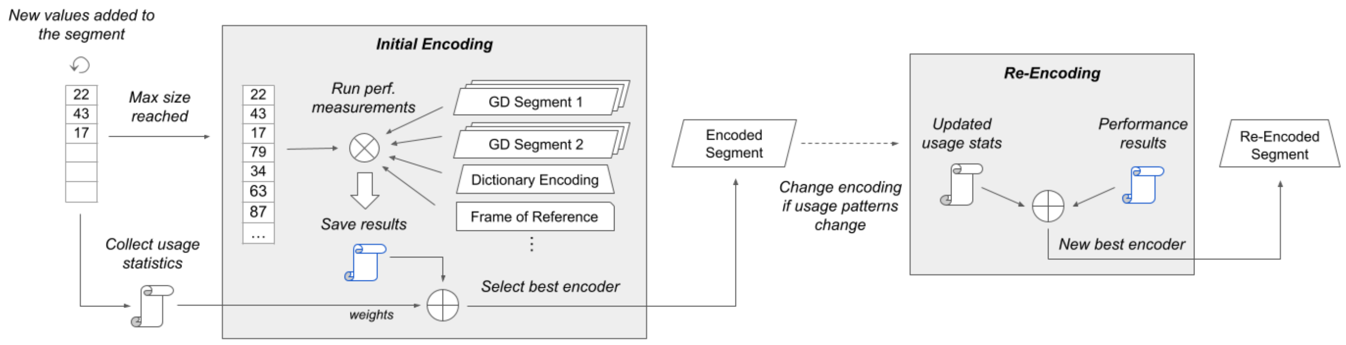


Figure 7: Proposed adaptive segment encoding process for autonomous databases.

which describes a system that automates maintenance and optimization tasks. If a database has up-to-date information about the usage patterns of each segment, it can assess whether the current encoding is still the best for the experienced workload. By running the performance measurements we proposed for finding the best GD Segment configuration, the database could compare every available encoder for every segment. In systems where compressed segment contents are immutable, the performance metrics stay valid throughout their whole lifetime. If usage patterns change over time, for example when data loses relevance and regular table scans are replaced by aggregate computations, the database can re-use the stored performance measurements together with the latest usage statistics in order to determine the best encoding going forward. It can also estimate the gains the new best encoding would yield over the current one, and decide whether it is worth re-encoding. This iterative process is illustrated by Figure 7.

The evaluation of encoders can be performed at regular time intervals, or triggered when a new segment of a column is completed. When the system concludes that a segment is worth re-encoding, it can proceed immediately or schedule the job for a later time when the user-facing workload is expected to be low. Even though segment encoding is assumed to be a background operation, measuring different configurations for GD Segment (and presumably other encoders too) is computationally expensive and might indirectly affect system performance. Therefore, this initial profiling could also be deferred to a quiet period, if possible.

It is worth noting, that the best balance between compression and access/scan speeds cannot be derived automatically from the proposed performance metrics and usage statistics, but it is rather an arbitrary choice. The database user, administrator or system itself must decide how much speed they are willing to trade for a smaller memory footprint and, hence, a lower cost. Sometimes the same segment encoder configuration yields the best compression and highest speed at the same time. In this case the decision is trivial, but many times it is less so. For example, when a segment is used extremely rarely, search and access performance is much less important than size reduction, thus a heavy compressor like LZ4 is a better option than dictionary encoding, even though it has orders of magnitude worse operational scores. The same choice may be present between different configurations of the same segment encoder.

5 EVALUATIONS

We evaluated GD Segment against uncompressed, dictionary encoded, frame-of-reference encoded and LZ4 compressed segments in two scenarios. First, we measured their compression and query performance in isolation using synthetic datasets. Then we integrated GD Segment 1 into Hyrise and measured the TPC-H and Join Order Benchmarks. The purpose of these tests is twofold. First, to see how GD Segment variants compare to commonly used segment encoders on typical integer columns. We also wanted to see how important it is to select the deviation size based on the data and relative importance of different factors, e.g., what the performance gain is when the deviation size is chosen based on measurements versus constant 8 bits.

5.1 Standalone Evaluation

We have implemented dictionary, Frame-of-Reference and LZ4[3] segment encoders in C++17 for standalone testing. They are functionally equivalent to their Hyrise counterparts: dictionary encoder uses a byte-aligned attribute vector and the same custom table scan logic, PFoR encodes the segment in blocks of 2048 values. The only difference in our implementation is that LZ4 segment compresses the whole segment as a single block (while in Hyrise it partitions the data to 16kB blocks), therefore a full segment decompression is required in the beginning of random access tests. Both LZ4 and PFoR segments fully decompress the values during table scans. GD Segment variants measure compression and query performance with all deviation sizes between 1 and 30 bits. We used the default segment size of Hyrise, which is 65535 elements. All standalone measurements were performed on a single thread of a 2.6 GHz Intel Core i7 CPU with 256 kB L2 and 12 MB L3 cache. The segments are stored and profiled entirely in memory.

We used the same synthetic datasets as Heinzl *et al.* in [12] with the addition of a primary key segment. The datasets are: (i) uniformly distributed random numbers between 0 and 2^{32}, (ii) sorted equidistant numbers with the step of 5, (iii) years between 1900 and 2100, (iv) months between 1 and 12, (v) a time series (power consumption readings of a household) starting at 10^6 and (vi) primary key starting with 1.

Compression gain is reported as a percentage of decrease in size, e.g., 0% indicates no compression and 50% means the compressed

[3]Using the ZLIB library, which is natively available in every operating system

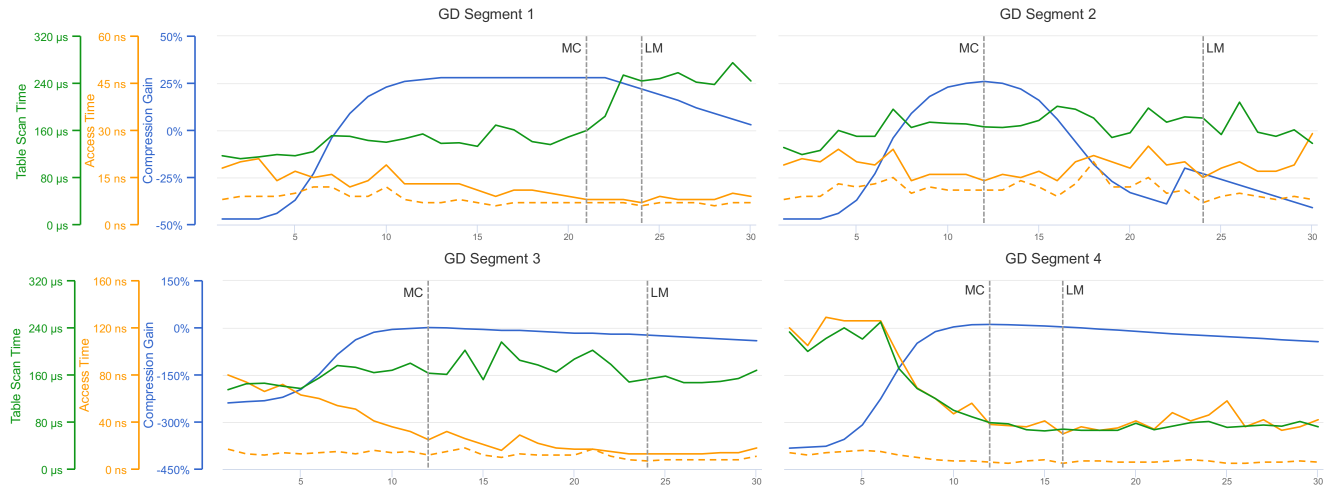


Figure 8: Compression gain (higher is better), access (dashed line: sequential, solid: random) and table scan times (lower is better) of the time series segment encoded with every GD Segment variant at the whole range of deviation sizes, between 1-30 bits (x axis). The best deviation size of Maximum Compression (MC) and Late Materialization (LM) performance presets are marked. Note, that the access time and compression scales are different for GD Segments 1-2 and 3-4.

segment is half the size of the original data vector. To measure random access time, we dereferenced a uniform random set of 6553 offsets (10% of the segment size), and report the average time. For sequential access, all offsets from 0 to 65535 are requested and the average time is reported. Segments that do not support random access (LZ4 and GD Segment 4) decompress the whole segment and return elements of the reconstructed integer vector. Decompression time is included in the reported times for these two segment types. For table scans we generate a uniform random set of 655 query values (1% of the segment size) in the range of the stored values with an extension of ± 10% to simulate out-of-range scans. Then, we perform a table scan with all six predicates for every query value and report the average time of all 3930 scans.

Note, that it would be straightforward to differentiate table scan times per predicate as separate performance metrics and allow the database to assign different weights to each one when determining the best encoding for a segment. We chose to aggregate all predicates only to ease the visualization of our results by decreasing the number of dimensions. We are also not reporting compression and decompression times, because they are irrelevant in a database where both initial encoding and re-encoding are non-blocking background operations. Therefore, encoding speed does not directly affect the user-perceived query performance. In use-cases where segment encoding is done synchronously, these should be part of the segment performance report as well.

Figure 8 illustrates how each metric changes when different deviation sizes are used to encode the time series segment. As a general trend it can be stated that when compression gain increases, both access and table scan times decrease. However, the deviation size that maximizes compression (marked MC) and expected performance of late materialization databases (marked LM) are different in every segment variant for this column. Therefore, we cannot assume that maximum compression automatically results in the best query performance.

It is also ill-advised to assign a static default deviation size to each GD Segment variant, since their behavior changes significantly with different data distributions. Figure 9 shows the measured metrics of GD Segment 1 across all deviation sizes, encoding different columns. Again, the configuration that yields the highest compression (MC) is different than the best deviation size for late materialization. Furthermore, we cannot predict which size is best for different configuration of weights or how they compare to each other (e.g., the best deviation for maximum compression is often smaller than the one for late materialization). For GD segments, the concrete data determines performance factors, therefore running the diagnostics cannot be skipped if we want to optimize for a certain goal.

Table 3 and Table 4 lists the complete result set of encoding the six synthetic datasets with different segments. For GD Segments, we report the best deviation size for each of four weight distributions shown on Figure 6, including Early Materialization (EM) and Equal weights (EQ) in addition to MC and LM. We make the following observations.

Dictionary Encoding significantly inflates the data when all values are different, but it still achieves consistently low access speeds due to the very few memory accesses and simple algorithm for dereferencing an offset. It excels at datasets with low cardinality, achieving very high compression and speed at the same time. Patched Frame-of-Reference is the fastest in data access in almost every case, and its size reduction is also among the best for most data distributions. An excellent choice as the default compressor in column stores. The table scan performance of GD Segment 4 is the best across all encoders in nearly every segment, since it is heavily optimized for this operation, at the expense of access speed and compression. GD Segment 1 achieves the highest compression (when optimized for this metric alone) in 3 out of 6 segments. Its

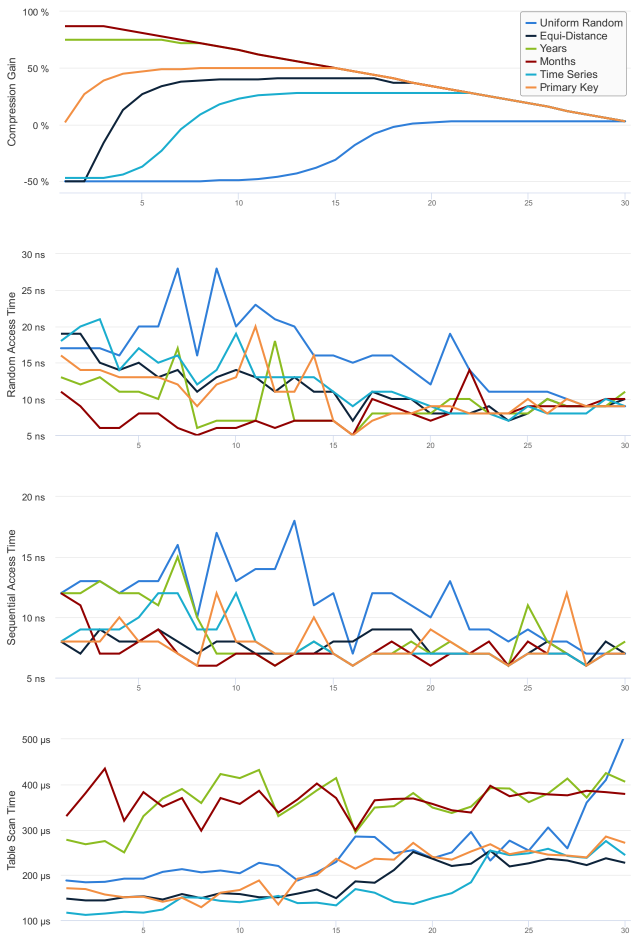


Figure 9: Performance metrics of GD Segment 1 at different deviation sizes (x axis) on all six synthetic datasets.

random and sequential access performance are the best among different GD Segment variants, since it requires the fewest memory accesses during dereferencing. Its table scan performance is very similar to Dictionary Encoding with almost all data distributions. LZ4 achieves very high compression levels (unless presented with the notoriously uncompressible uniform random data), but it lacks efficient random access and fast table scans. This method is a reasonable default for segments that are very rarely accessed.

Note, that LZ4 and GD Segment 4 appear to have adequate sequential access speeds, but this is only a side-effect of our measurement methodology. Since these segment types do not support constant-time random or sequential access, a full decompression is performed at the beginning of the access tests and measurements read the decompressed regular vector. Therefore, the cost of the expensive one-time decompression is amortized by the 65535 extremely fast direct memory accesses. As a result, the sequential access column of GD Segment 4 and LZ4 practically only shows their decompression speeds.

5.2 Benchmarks in Hyrise

To observe the real-world performance of GD Segments and see the effects of different selection criteria for the best deviation size, we implemented GD Segment 1 as a segment encoder in Hyrise, and ran two industry-standard benchmarks: TPC-H and Join Order Benchmark (JOB). We chose Hyrise for this evaluation because it satisfies all our expectations about handling segments:

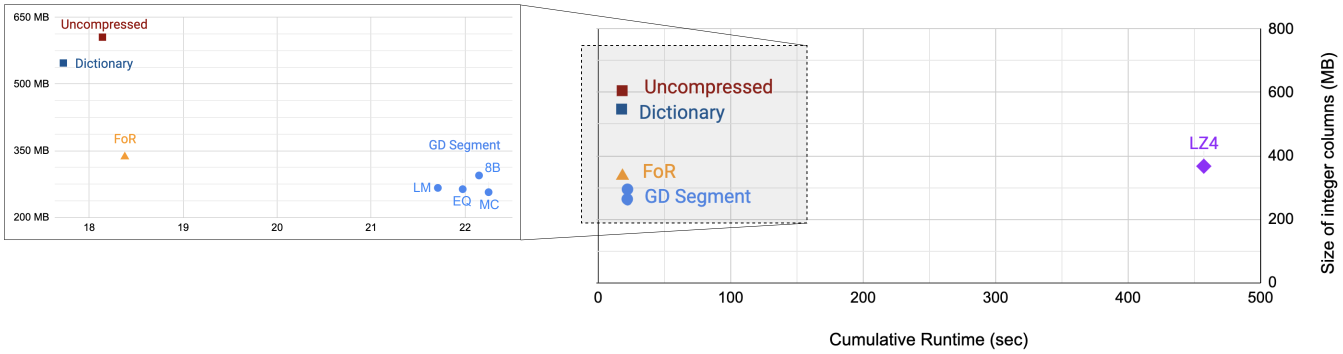
- Segments store values of a single type.
- Segment encoding is a non-blocking background operation.
- Encoded segments are immutable.
- New segment encoders can be added relatively easily via an extensible framework.

Hyrise is a late materialization database where lists of segment offsets are passed between query nodes, and iterators are used as the interface between the query engine and segment encoders. As a result, segment performance (e.g., query performance with a given segment type) is mainly determined by the speed of iterator dereferencing. We decided to integrate GD Segment version 1, because it has the best random and sequential access performance across the four variants. When a GD segment is first encoded, we run all performance measurements described earlier and store the results in a local JSON file. Subsequent encodings of the same segment (e.g., running TPC-H again) simply load the metrics from the disk instead of having to run all tests again. The relative weights of compression and speeds are also read from a local configuration file and used by the encoder on the fly when selecting the deviation size of the segment. We measure different weight distributions for GD segments by changing the weights in the config file and re-running the benchmark. The configurations tested in the benchmarks and marked on the figures are the following: fixed 8-bit deviation size (*8B*), Late Materialization (*LM*), Maximum Compression (*MC*) and Equal Weights (*EQ*).

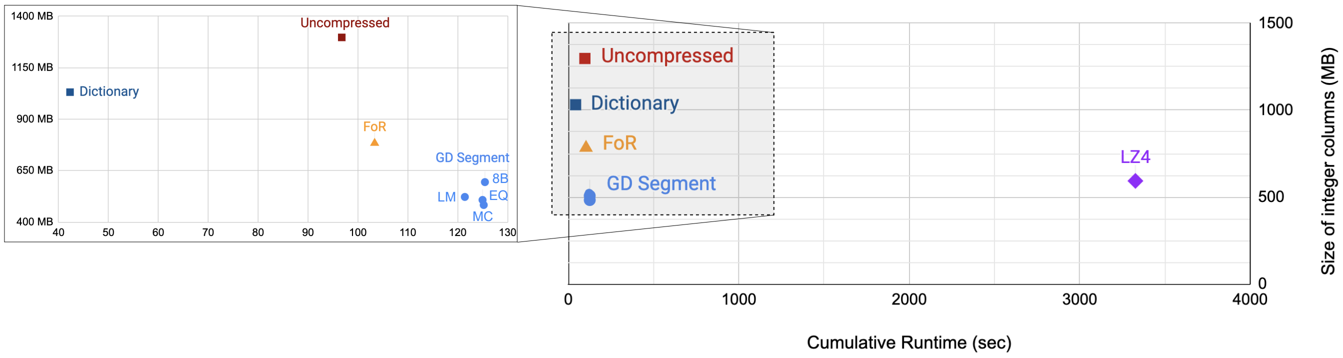
Note, that the built-in Frame-of-Reference encoder in Hyrise actually implements the PFoR algorithm with a block size of 2048, but it is called FoR encoder in the source code. Therefore, we will also refer to it the same way in our evaluations. Columns that cannot be encoded with the tested encoder (e.g., float and string columns with FoR and GD) are left uncoded. We report the total compressed size of integer columns and the sum of average query execution times as the cumulative benchmark runtime. All of the computation done for these benchmarks was performed on the UCloud[4] interactive HPC system, which is managed by the eScience Center at the University of Southern Denmark. All measurements were performed in single-threaded mode.

Figure 10 shows the achieved compression and runtime of TPC-H at scale factor 5 and Join Order Benchmark. The two results show similar patterns of encoder performance. Dictionary encoding is the worst compressor in both benchmarks, achieving only 10% reduction in TPC-H and 21% in JOB, but it is the fastest as well. According to our measurements, 2% faster than unencoded segments in TPC-H and 56% faster in Join Order Benchmark. This makes Dictionary Segment the only contender for Hyrise that simultaneously decreases data size and makes queries faster.

[4]https://docs.cloud.sdu.dk



(a) TPC-H (scale factor 5)



(b) Join Order Benchmark

Figure 10: Size of integer columns and total runtimes of analytical benchmarks in Hyrise, with different segment encodings.

However, if the query speed with unencoded segments is satisfactory for a certain use-case and decreasing the memory footprint is more important, the other three encoders offer better compression with different levels of performance penalty. Frame-of-Reference achieves 44% (TPC-H) and 40% (JOB) size reductions of integer columns for only 1% and 7% higher query times. Traditionally the only other option was to almost completely sacrifice query performance for similar or slightly better compression, using LZ4. It reduces the memory consumption by 39% (TPC-H) and 54% (JOB) for a staggering 24x-33x decline in query speed. This performance makes LZ4 a viable option only on rarely queried columns.

GD Segments provide a new trade-off between query performance and compression, which is close to the speed of FoR with better size reduction than LZ4. In TPC-H, GD segment variants achieve compression between 51% and 58%, the highest among all encoders and 5-6x better than Dictionary encoding. In terms of query performance, GD is 20-23% worse than unencoded segments, depending on the relative importance of metrics that guided the deviation size selection. Results for GD are similar in the Join Order Benchmark as well, with slightly higher compression (54-63%) and lower performance (26-30%). The different weight configurations used to determine the deviation size work as intended. The *Late Materialization* setting yielded the fastest GD Segment in both benchmarks, while *Maximum Compression* is indeed the one with the lowest total memory consumption. As expected, the fixed 8-bit deviation is worse than both of them.

6 CONCLUSIONS AND FUTURE WORK

In this paper we have introduced a new column compression family based on generalized deduplication for integer sequences, and four practical designs for segment encoders in columnar databases. They aim to optimize for both memory footprint reduction and efficient query execution, without having to decompress the whole segment. We have shown that the performance of the proposed segments is comparable to current state-of-the-art encoders, offering a new trade-off point between compression and query speed. Additionally, we proposed an adaptive segment encoder selection scheme for autonomous databases, based on the same diagnostic and evaluation mechanism we use to automatically select the best configuration for our segment.

Our future plans include enhancing the storage layer of Hyrise with i) the ability to collect detailed, segment-level usage statistics, including the frequency and predicate of table scans, ii) a standardized set of performance measurements to evaluate segment encoders at their full parameter space, and iii) an encoding selection framework that combines the experienced usage patterns of segments with the performance metrics of potential encoders to find the best possible one, and re-evaluates this decision when the usage patterns sufficiently change.

Table 3: Detailed results of standalone evaluation of GD Segments. Values in bold indicate the best of the column. MC, EQ, LM and EM in the *Best Deviation Size* (Best Dev.) column marks which segment encoding (and which deviation size in case of GD) yields the best overall choice for the given optimization goal.

		Uniform Random					Sorted Equi-Distance				
	Version	**Best Dev.**	**Comp.**	**Rand. Acc**	**Seq. Acc**	**Scan**	**Dev.**	**Comp.**	**Rand. Acc**	**Seq. Acc**	**Scan**
GD S1	MC	28 (MC)	**3%**	14 ns	10 ns	483 μs	17	41%	8 ns	7 ns	183 μs
	EQ	24	**3%**	13 ns	11 ns	359 μs	16	41%	6 ns	6 ns	182 μs
	LM										
	EM										
GD S2	MC	13	-48%	31 ns	21 ns	353 μs	10	40%	19 ns	12 ns	228 μs
	EQ	16	-50%	19 ns	11 ns	369 μs	8	39%	15 ns	10 ns	225 μs
	LM										
	EM						1	-50%	24 ns	9 ns	151 μs
GD S3	MC	20	-20%	35 ns	23 ns	346 μs	7	20%	30 ns	14 ns	201 μs
	EQ	23	-25%	31 ns	23 ns	311 μs	18	-13%	14 ns	10 ns	187 μs
	LM	24	-28%	29 ns	20 ns	352 μs					
	EM	23 (EM)	-25%	30 ns	23 ns	311 μs	13	6%	22 ns	11 ns	173 μs
GD S4	MC	21	-19%	50 ns	7 ns	89 μs	8	23%	34 ns	6 ns	71 μs
	EQ	25	-28%	47 ns	6 ns	76 μs	26 (EM)	-30%	33 ns	5 ns	**68 μs**
	LM										
	EM	26	-31%	47 ns	6 ns	**75 μs**					
Dictionary			-50%	8 ns	7 ns	421 μs		-100%	8 ns	**4 ns**	189 μs
FoR			0%	**7 ns**	6 ns	476 μs	EQ, LM	50%	**4 ns**	**4 ns**	301 μs
LZ4		EQ, LM	0%	10 ns	**3 ns**	262 μs	MC	**65%**	112 ns	13 ns	779 μs

		Years					Months				
	Version	**Best Dev.**	**Comp.**	**Rand. Acc**	**Seq. Acc**	**Scan**	**Dev.**	**Comp.**	**Rand. Acc**	**Seq. Acc**	**Scan**
GD S1	MC	6 (MC)	**75%**	26 ns	20 ns	408 μs	3 (MC)	**87%**	6 ns	6 ns	415 μs
	EQ	16	47%	5 ns	6 ns	314 μs	4	84%	6 ns	6 ns	292 μs
	LM						8	72%	5 ns	6 ns	297 μs
	EM						4	84%	6 ns	6 ns	292 μs
GD S2	MC	5	75%	12 ns	13 ns	272 μs	3	87%	6 ns	6 ns	328 μs
	EQ	16	75%	6 ns	6 ns	257 μs	4	87%	6 ns	6 ns	232 μs
	LM										
	EM										
GD S3	MC	4	75%	14 ns	14 ns	290 μs	4 (EQ, EM)	87%	5 ns	6 ns	**231 μs**
	EQ	16	75%	6 ns	6 ns	244 μs					
	LM										
	EM										
GD S4	MC	1	46%	32 ns	6 ns	185 μs	1	47%	39 ns	5 ns	295 μs
	EQ	2	44%	31 ns	5 ns	**182 μs**					
	LM	8	25%	29 ns	5 ns	319 μs	16	0%	29 ns	5 ns	343 μs
	EM	2	44%	31 ns	5 ns	**182 μs**	1	47%	39 ns	5 ns	295 μs
Dictionary		EQ, LM, EM	75%	**4 ns**	**4 ns**	319 μs	LM	75%	**4 ns**	**4 ns**	345 μs
FoR			75%	**4 ns**	5 ns	354 μs		75%	10 ns	4 ns	368 μs
LZ4			65%	97 ns	12 ns	711 μs		84%	66 ns	8 ns	535 μs

7 ACKNOWLEDGEMENT

This work was partially financed by the SCALE-IoT project (Grant No. DFF - 7026-00042B) granted by the Danish Council for Independent Research, the Aarhus Universitets Forskningsfond Starting Grant Project AUFF-2017-FLS-7-1, and Aarhus University's DIGIT Centre.

REFERENCES

[1] [n.d.]. LZ4. Available at https://lz4.github.io/lz4 (2022/06/21).
[2] [n.d.]. TPC-H Benchmark Website. Available at https://www.tpc.org/tpch (2022/06/21).
[3] Daniel Abadi, Samuel Madden, and Miguel Ferreira. 2006. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*. 671–682.
[4] Martin Boissier. 2021. Robust and Budget-Constrained Encoding Configurations for in-Memory Database Systems. *Proc. VLDB Endow.* 15, 4 (dec 2021), 780–793.

Table 4: Detailed results of standalone evaluation of GD Segments on Time Series and Primary Key synthetic datasets (continuation of 3)

		Time Series					Primary Key				
	Version	Best Dev.	Comp.	Rand. Acc	Seq. Acc	Scan	Dev.	Comp.	Rand. Acc	Seq. Acc	Scan
GD S1	MC	21	28%	8 ns	7 ns	168 μs	15	50%	10 ns	9 ns	231 μs
	EQ						8	50%	9 ns	7 ns	139 μs
	LM	16	28%	7 ns	6 ns	167 μs	16	47%	6 ns	6 ns	238 μs
	EM						8	50%	9 ns	7 ns	139 μs
GD S2	MC	12	26%	18 ns	14 ns	200 μs	9	50%	13 ns	8 ns	145 μs
	EQ	16	6%	11 ns	7 ns	189 μs	8	50%	10 ns	7 ns	159 μs
	LM										
	EM	3	-47%	21 ns	9 ns	129 μs	3	39%	14 ns	8 ns	131 μs
GD S3	MC	12	1%	24 ns	12 ns	185 μs	4	32%	24 ns	11 ns	201 μs
	EQ										
	LM	23	-20%	13 ns	8 ns	146 μs	18	-6%	11 ns	7 ns	171 μs
	EM										
GD S4	MC	12	11%	46 ns	6 ns	73 μs	6	33%	41 ns	7 ns	**79 μs**
	EQ						8 (EM)	28%	35 ns	5 ns	84 μs
	LM	16 (EQ,LM,EM)	3%	32 ns	**4 ns**	**69 μs**	16	0%	28 ns	5 ns	103 μs
	EM						8 (EM)	28%	35 ns	5 ns	84 μs
Dictionary			-47%	6 ns	4 ns	145 μs		-100%	7 ns	4 ns	202 μs
FoR			0%	**5 ns**	4 ns	310 μs	LM	50%	**4 ns**	**4 ns**	275 μs
LZ4		MC	**57%**	180 ns	20 ns	103 μs	MC	**65%**	110 ns	13 ns	744 μs

https://doi.org/10.14778/3503585.3503588

[5] Lujing Cen, Andreas Kipf, Ryan Marcus, and Tim Kraska. 2021. LEA: A Learned Encoding Advisor for Column Stores. In *Fourth Workshop in Exploiting AI Techniques for Data Management* (Virtual Event, China) *(aiDM '21)*. Association for Computing Machinery, New York, NY, USA, 32–35. https://doi.org/10.1145/3464509.3464885

[6] 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 *EDBT*. 313–324.

[7] Franz Faerber, 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

[8] Franz Färber, Norman May, Wolfgang Lehner, Philippe Grosse, Ingo Müller, Hannes Rauhe, and Jonathan Dees. 2012. The SAP HANA Database – An Architecture Overview. *IEEE Data Eng. Bull.* 35 (2012), 28–33.

[9] Marcell Fehér, Niloofar Yazdani, Morten Tranberg Hansen, Flemming Enevold Vester, and Daniel Enrique Lucani Rötter. 2020. Smart Meter Data Compression using Generalized Deduplication. In *GLOBECOM 2020 - 2020 IEEE Global Communications Conference*. IEEE. https://doi.org/10.1109/GLOBECOM42002.2020.9322393 2020 IEEE Global Communications Conference, Globecom ; Conference date: 07-12-2020 Through 11-12-2020.

[10] Florian Funke, Alfons Kemper, and Thomas Neumann. 2012. Compacting Transactional Data in Hybrid OLTP & OLAP Databases. *Proc. VLDB Endow.* 5, 11 (jul 2012), 1424–1435. https://doi.org/10.14778/2350229.2350258

[11] Jonathan Goldstein, Raghu Ramakrishnan, and Uri Shaft. 1998. Compressing relations and indexes. In *Proceedings 14th International Conference on Data Engineering*. IEEE, 370–379.

[12] Linus Heinzl, Ben Hurdelhey, Martin Boissier, Michael Perscheid, and Hasso Plattner. 2021. Evaluating Lightweight Integer Compression Algorithms in Column-Oriented In-Memory DBMS.. In *ADMS@ VLDB*. 26–36.

[13] 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 *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 843–856. https://doi.org/10.1145/3448016.3457283

[14] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proceedings of the VLDB Endowment* 9, 3 (2015), 204–215.

[15] Lin Ma. 2021. *Self-Driving Database Management Systems: Forecasting, Modeling, and Planning.* Ph.D. Dissertation. Carnegie Mellon University.

[16] Lin Ma, Dana Van Aken, Ahmed Hefny, Gustavo Mezerhane, Andrew Pavlo, and Geoffrey J Gordon. 2018. Query-based workload forecasting for self-driving database management systems. In *Proceedings of the 2018 International Conference on Management of Data*. 631–645.

[17] 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.

[18] Lars Nielsen, Dorian Burihabwa, Valerio Schiavoni, Pascal Felber, and Daniel E. Lucani. 2021. MinervaFS: A User-Space File System for Generalised Deduplication: (Practical experience report). In *2021 40th International Symposium on Reliable Distributed Systems (SRDS)*. 254–264. https://doi.org/10.1109/SRDS53918.2021.00033

[19] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C Mowry, Matthew Perron, Ian Quah, et al. 2017. Self-Driving Database Management Systems.. In *CIDR*, Vol. 4. 1.

[20] Mark Raasveldt and Hannes Mühleisen. 2020. Data Management for Data Science - Towards Embedded Analytics. In *CIDR*.

[21] Rasmus Vestergaard, Daniel E Lucani, and Qi Zhang. 2019. Generalized Deduplication: Lossless Compression for Large Amounts of Small IoT Data. In *European Wireless Conference*. VDE, 1–5.

[22] Adrian Vogelsgesang, Michael Haubenschild, Jan Finis, Alfons Kemper, Viktor Leis, Tobias Muehlbauer, Thomas Neumann, and Manuel Then. 2018. Get Real: How Benchmarks Fail to Represent the Real World. In *Proceedings of the Workshop on Testing Database Systems* (Houston, TX, USA) *(DBTest'18)*. Association for Computing Machinery, New York, NY, USA, Article 1, 6 pages. https://doi.org/10.1145/3209950.3209952

[23] 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, Article 15 (mar 2015), 28 pages. https://doi.org/10.1145/2735629