# Can Delta Compete with Frame-of-Reference for Lightweight Integer Compression?

Julia Spindler
Technical University of Munich
julia.spindler@tum.de

Philipp Fent
CedarDB
philipp@cedardb.com

Adrian Riedl
Technical University of Munich
adrian.riedl@in.tum.de

Thomas Neumann
Technical University of Munich
neumann@in.tum.de

## ABSTRACT

Compressing data in a columnar layout has large benefits for storage size. Lightweight compression schemes offer quick compression and allow execution directly on compressed data, while they do not compress aggressively. Delta encoding is a promising candidate for integer compression, especially for ID columns, where traditional lightweight compression achieves only low compression ratios. In this paper, we show that delta encoding can achieve a 4× higher compression ratio compared to other lightweight compression schemes. While delta compression performs similarly to other schemes in unpredicated scans, it struggles in selective scans, even with optimizations that allow for worse compression. Therefore, we implement a new version of frame-of-reference encoding that combines the strengths of both delta and FOR encoding. This approach matches the compression ratio of delta encoding, surpasses delta in all decompression metrics, and is up to 23% faster compared to the standard compression scheme implementations.

**VLDB Workshop Reference Format:**
Julia Spindler, Philipp Fent, Adrian Riedl, and Thomas Neumann. Can Delta Compete with Frame-of-Reference for Lightweight Integer Compression?. VLDB 2024 Workshop: Fifteenth International Workshop on Accelerating Analytics and Data Management Systems Using Modern Processor and Storage Architectures(ADMS 2024).

## 1 INTRODUCTION

Database systems use compression for data storage to reduce the size of data that needs to be processed and minimize processing times. Data movement is already one of the most expensive parts of data centers [5, 17]. Therefore, many systems, such as BigTable [6], Oracle [16], or Snowflake [7], employ data compression on the data stored in a database. Data compression is not only relevant for storage but also beneficial for in-memory operations, particularly where the memory-bandwidth often limits vectorized table scans. Thus, main-memory oriented systems like DuckDB [18], HyPer [11], or Hyrise [9] implement lightweight compression techniques, which allow operations directly on compressed data. This lightweight compression not only saves storage space, but also increases performance, since it reduces memory traffic, and increases the effective capacity of data that can be kept in-memory.

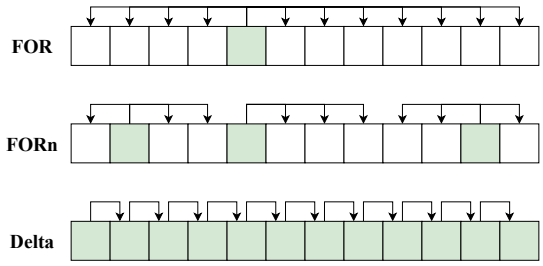


**Figure 1: Three concepts to compress small range integer data: Frame-of-reference (`FOR`) uses one reference point for a whole DataBlock; `FORn` uses one reference point per n values; `Delta` uses each value as a reference point for the next.**

Currently used techniques for lightweight compression include bit-packing, frame-of-reference encoding, or dictionary compression. These techniques work well and strike a balance between fast decompression speeds and high compression ratios [3]. However, they can still be suboptimal for certain cases. To cover more cases, Afroozeh and Boncz [4] proposed an efficient delta compression technique that can potentially enable run-length encoding of similar deltas. Delta compression is especially effective for integer ID columns, where the values span a large range, but are usually ascending with very similar deltas. This compression technique stores only the differences between consecutive values, which can then be compactly bit-packed. However, delta compression lacks efficient random access properties, which poses difficulties when integrating it into a database system.

In the following, we first analyze the implementation of columnar integer compression techniques and how they integrate in our research database system Umbra [14]. Next, we adapt the FastLanes delta compression technique, using small baseline strides to support reasonably efficient random access and query processing. Lastly, we apply this small-stride adaptation to frame-of-reference encoding, which then achieves similar compression ratios with a simpler and faster implementation.

**Compression schemes**. Umbra organizes tuples in columnar blocks of $2^{16}$ tuples. When enough tuples are put into a relation,


Proceedings of the VLDB Endowment. ISSN 2150-8097.

Umbra selects the best compression for this particular block before encoding the data.

*Single* compression is used if the integer in a block is the same for each tuple. For instance, this occurs when an integer represents a status code where one value predominates.

*Increment* compression is helpful for IDs, as they usually start at 1 and are then steadily incremented. If this is known, this compression stores the first ID in the block, and the other values can be retrieved using their indices.

*Truncation and frame-of-reference (FOR)* compression make use of the fact that while SQL integers are typically 4 or 8 bytes, values within a block might need fewer bits. To avoid complexity, truncation happens only on the 1, 2, or 4-byte level.

*Frame-of-reference* (FOR) works by identifying the minimum value in a block and storing each subsequent value as a positive difference from this minimum. This method is especially effective for keys with a narrow range of values, which results in much smaller values after subtracting the minimum.

*Dictionary* compression, on the other hand, benefits from data with many duplicates. This can be the case for integer columns having only minimal domain sizes, such as status codes, or foreign keys.

**Analysis of schemes**. The implemented compression schemes are already quite versatile. We evaluate them on the TPC-H and JOB datasets to analyze their effectiveness. One compression scheme not currently employed is *delta encoding*, which stores only the differences between consecutive values. These deltas are typically smaller than the original values and can benefit from byte truncation. We identified columns in both datasets that would benefit from delta encoding. We compared the effectiveness of the compression across 7 integer columns using the current compression strategies, delta compression, and compression using the `xz` command line utility as a baseline. The upper bound and reference value is storing the values uncompressed. As shown in Table 1, delta compression sometimes achieves compression where columns would otherwise remain uncompressed or could be truncated to 1 byte instead of 2 bytes. For example, in both the `order` and `lineitem` relations, the `orderkey` column is sorted in ascending order with gaps between values. Therefore, the range of the values in a $2^{16}$ block is larger than that of a 2-byte integer, meaning Umbra stores these columns uncompressed. Since the delta between values never exceeds 127, the column can be compressed into 1-byte value deltas. In contrast, in the case of the keys in the `name` and `keyword` tables, the values are distributed more randomly, which can be seen in the low `xz` compression ratios. Nevertheless, there is still a connection between an index of a value and its position in the column, though, as the values tend to increase. This allows for effective 2-byte truncation of deltas between values.

**Delta Compression**. Incorporating delta compression into the database system could reduce memory consumption by up to 73% for the analyzed ID columns. Since delta encoding stores the delta to the last preceding value, this generally results in smaller deltas compared to FOR encoding. Delta compression has the advantage of being able to compress integers over a large range if the differences between those values stay small. This is something that can happen in datasets that are initially incremental IDs, but these can get disrupted when entries are deleted or moved around.

**Table 1: Compression ratios of uncompressed data compared to data compressed with and without delta compression and the xz utility for different columns of the JOB and TPC-H datasets.**

| | | max size | no delta | delta | xz |
|---|---|---|---|---|---|
| TPC-H | o_orderkey | 6 MB | 1.00 | 3.70 | 14.3 |
| | l_orderkey | 24 MB | 1.61 | 3.70 | 25.0 |
| | ps_partkey | 3.2 MB | 2.00 | 3.70 | 50.0 |
| JOB | cast_info id | 145 MB | 2.00 | 2.08 | 14.3 |
| | movie_info id | 59 MB | 2.33 | 2.63 | 14.3 |
| | name id | 17 MB | 1.0 | 1.85 | 2.44 |
| | keyword id | 0.55 MB | 1.02 | 1.92 | 2.27 |

**Challenges**. Delta encoding introduces dependencies between values, complicating parallelization and the evaluation of predicates on compressed data. These challenges make delta encoding more complex than the currently employed schemes.

In this work, we implement delta compression into the Umbra database system [14]. Our scheme compresses 4- or 8-byte integer columns into 1- or 2-byte delta-encoded columns. First, we provide an overview of compression in contemporary database systems in Section 2. Then, we introduce our implementation and layout for delta compression in Section 3, including the use of SIMD for decompression. We adapt similar techniques for FOR encoding in Section 4. Lastly, we evaluate our approaches in Section 5.

## 2 RELATED WORK

Most database systems incorporate some form of compression. Oracle Database and SAP HANA primarily utilize dictionary encoding [2, 16]. While SAP HANA includes stronger compression schemes, it does not implement delta encoding. Vectorwise implements dictionary, frame-of-reference, and delta encoding for columnar compression [19]. The delta compression uses arbitrary bit widths; all decompression is only done on a whole block of compressed data. The widely used columnar storage format Parquet also combines delta encoding with frame-of-reference encoding to ensure positive deltas [1]. `BTRBLOCKS` employs blockwise columnar compression on any type of data, combining various compression schemes iteratively on potentially already compressed data [10]. Decompression uses SIMD to speed up the execution. Unlike our approach, which always selects the optimal compression methods, `BTRBLOCKS` relies on heuristics and data sampling and does not include delta encoding in its compression schemes.

In their paper on the FastLanes compression layout, the authors propose SIMD approaches for several compression schemes, including delta encoding [4]. This method leverages compiler auto-vectorization, packing delta values into bits, whereas we use simpler byte-packing. However, this approach is not integrated into any database system, meaning predicate evaluation and random access on the data are not addressed. Additionally, the order of values is not preserved during decompression, which could lead to issues in practical systems.

Umbra currently utilizes several lightweight compression schemes on its Data Blocks [11]. These Data Blocks are compressed using the optimal scheme and include a header containing the minimum and maximum values of the column. This metadata allows the database engine to skip entire blocks when executing SARG-able predicates. Scans on compressed data are vectorized and then fed into the JIT-compiled query pipeline. However, delta encoding has not been implemented.

In [8], the authors explore a method for compressing integer columns that uses the first bits of the integer as a base and the remaining bits as deviations. The bases are stored as a dictionary without duplicates. This approach is similar to delta encoding, as it is most effective when neighboring values are similar. However, it introduces the base bit-width as a new parameter that should ideally be configured at runtime.

## 3 DELTA COMPRESSION

In the following section, we will describe the implementation of delta encoding in our database system Umbra. Compression in Umbra works as follows: if the user selects the Blocked Relation during creation, accumulated data is compressed into Data Blocks once reaching a threshold of $2^{16}$ tuples. This compression is chosen based on the highest compression ratio. The Data Blocks can be serialized to disk, requiring us to determine an appropriate serialization layout for delta compression.

Our implementation of delta encoding in Umbra supports two different types of decompression routines:

*Range-based*: This routine receives a start and end index to decompress all tuples within that range.

*Match-based*: This routine receives a match vector, constructed, for example, from a predicate evaluation and only decompresses tuples at the specified indices.

### 3.1 Data Layout

We need to include additional information in its header to store delta-encoded data. While theoretically, compressing a block of integers using delta compression only requires storing the first value of the delta-compressed column in the block header, this approach has two significant drawbacks:

*No Support for Thread- or Data-Level Parallelization*: This method impedes optimizations that parallelize data decompression. Each thread would need to start decompression from the beginning of the block, preventing the use of SIMD (Single Instruction, Multiple Data) for parallel processing.

*Inefficient Point Access*: Point access in delta compression can be highly inefficient, as the entire block must be decompressed even if only a single specific tuple is needed.

To overcome these problems, the block header stores multiple values, so-called data points, from the column before the deltas. The number of values in the header can be configured by varying the stride size parameter.

By default, the stride size is 1024, meaning the value of every $1024^{\text{th}}$ tuple is stored in front of the compressed data. This stride size corresponds to the morsel size of a thread in Umbra using the approach introduced by Leis et al. [12]. In an ideal scenario for a complete scan over the data, each thread would load the next data point from the header and start decompressing immediately. Storing the full-width value every 1024 tuples also keeps the memory overhead very low.

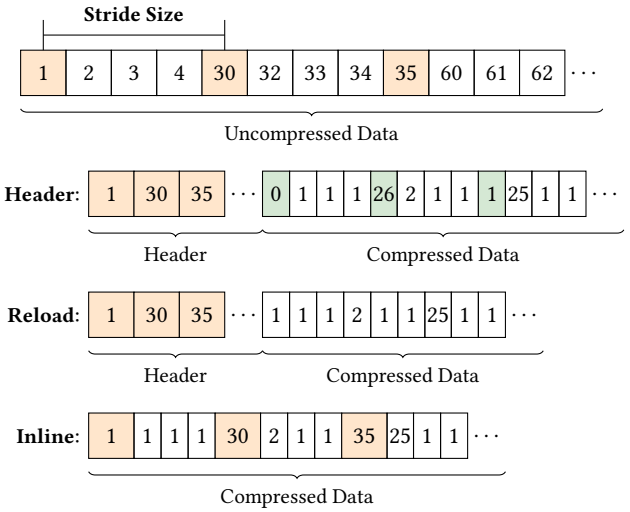


**Figure 2: Different storage layouts for delta compressed data with data points.**

In Figure 2, we present three different layouts to store delta-compressed data:

**Header**. For this layout, we store the delta values for all elements, including those already stored in the header. The advantage of this approach is that it simplifies the compression and decompression logic. However, it introduces redundancies: the values stored uncompressed in the header can be retrieved either directly from the header or by computing them using the delta values.

**Reload**. This approach is similar to the previous one but omits the redundant deltas. When accessing a value of an element, we either load it directly from the header if the element corresponds to a data point or perform the decompression logic for other elements.

**Inline**. Instead of storing the values of the data points in a dedicated header, they are stored inline with the deltas. Due to the uncompressed non-delta value being a larger data type compared to the delta values, we need an unaligned load to fetch it.

The latter two approaches require less storage compared to the first one. When $S$ is the integer data type that is used to store the delta, both approaches save $\frac{\text{numTuples}}{\text{strideSize}} \cdot \text{sizeof}(S)$ bytes. With our default stride size of 1024 and a block size of $2^{16}$ tuples, this results in 64 bytes less per data block. For example, the `l_orderkey` column would use 5824 bytes less for scale factor 1.

The downside of these approaches is that they generally do not support branchless decompression, as some indices in the compressed data need special handling.

In Figure 3, we compare all approaches on the `l_orderkey` attribute in the `lineitem` relation of the TPC-H scheme. We observe a drop in performance for the **Inline** and **Reload** approach due to the extra bookkeeping during decompression. However, this overhead becomes negligible during a selective scan using a match vector.

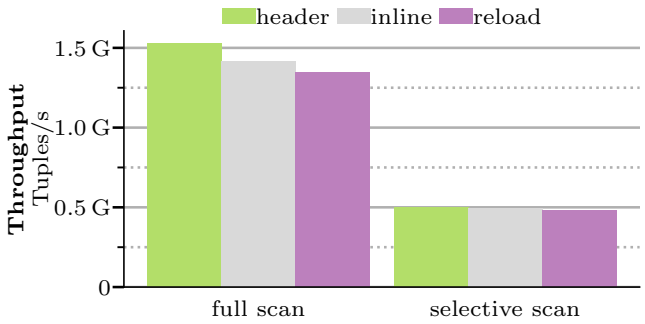


**Figure 3: Throughput for different layout approaches for a full scan and a 50% selective scan of `l_orderkey`.**

```
closest = r.begin / stride
prev = readerHeader[closest]
// align to r.begin
for (i = closest * stride + 1; i <= r.begin; ++i)
  prev += reader[i]
// start processing the elements of the range
*(writer++) = prev
for (i = r.begin + 1; i < r.end; ++i)
  prev += reader[i]
  *(writer++) = prev
```

**Listing 1: Decompression logic of delta-encoded data given the tuple range `r`.**

The overhead can be avoided by knowing that each thread stays within certain bounds. Adjusting the stride size accordingly makes the branch unnecessary since we only need to load one data point at the beginning of the decompression. In this scenario, all approaches achieve the same performance.

## 3.2 Delta Decompression

If the data is accessed in its compressed form, it needs to be decompressed before being passed along the data pipeline [13]. Decompression can be done on a consecutive range of tuples or using a match vector. This match vector contains the sorted indices of the tuples accessed in this Data Block. In either case, the decompression process begins by loading the closest preceding data point in the header from the index of the first tuple. The decompression loop then computes the values until reaching the end of the range or the last match. The implementation for a range can be seen in Listing 1.

Both implementations work branchfree, as any decompressed value is always written to the output buffer. In the case of a match vector, the pointer is only advanced if the index matches the current match, as seen in Listing 2.

## 3.3 Using SIMD

Database systems, especially those using vectorization, employ SIMD instructions to boost throughput [15]. We explore whether SIMD execution of delta decompression could enhance full scans over data, considering that other compression schemes in Umbra already utilize SIMD when feasible. With SIMD execution and delta compression, the problem of dependencies between the compressed values becomes even more pressing. It is now necessary to break

```
matchIndex = 0
closest = m[matchIndex] / stride
prev = readerHeader[closest]
for (i = closest * stride + 1; i < m.end; ++i)
  prev += reader[i]
  *writer = prev
  writer += i == m[matchIndex]
  matchIndex += i == m[matchIndex]
```

**Listing 2: Branchfree decompression logic of delta-encoded data given the match vector `m`.**

```
offsets = {7*stride, 6*stride, 5*stride, 4*stride,
           3*stride, 2*stride, stride  , 0}
for (i = r.begin, j = 0; i < r.end; i+=8, ++j)
  block = j / stride
  // store index of first value in register
  base = (j % stride) + block * 1024
  if (i % 1024 == 0)
    start = i / stride
    reg1 = load(readerDatapoint + start)
  reg2 = load(reader + iter)
  reg1 += reg2
  vindex = set1(base)
  vindex += offsets
  scatter(writer, vindex, reg1)
```

**Listing 3: Decompressing delta-encoded values using SIMD.**

dependencies not only between threads but also between SIMD lines. This problem is also discussed in the FastLanes paper [4].

Similar to their approach, we store uncompressed start values next to each other to load them into one register, ensuring each lane can independently begin decompression. To do this, we store these values in the header, just as with the scalar approach, but require more data points. Here, the data point stride size depends on a minimum tuple workload that each thread should perform and the size of the integer type $T$. The minimum workload is set to be 1024 tuples, corresponding to Umbra's morsel size. The data point stride $k$ is then given by $k = \frac{1024 \cdot \text{sizeof}(T)}{\text{sizeof(Register)}}$. We implement SIMD decompression using AVX512 registers, where sizeof(Register) = 64 bytes. The implementation of the SIMD decompression can be seen in Listing 3.

To make the decompression more efficient, the deltas are shuffled during compression instead of being stored in their original sequence. This method follows the approach detailed in the FastLanes paper [4]. The shuffled layout is illustrated in Figure 4. In this example, instead of storing the deltas the same as the elements from which they were extracted, the delta for the tuple at index 128 is stored next to the delta for the tuple at index 0 and so on. The first SIMD lane decompresses tuples at indices 0-127, the second lane handles indices 128-255, and so on. This setup allows for a simple load from the compressed data to start decompression, eliminating the need to gather start values from multiple locations. It is also important to note that at the beginning of every 1024 tuples, the first block of $\frac{n}{k}$ deltas are the deltas of values that have to be loaded from the header in any case and can, therefore, be omitted. In Figure 4, this would be the first eight values marked blue. Since we want to preserve the original order of the tuples after decompressing, the values are scattered into the output buffer.

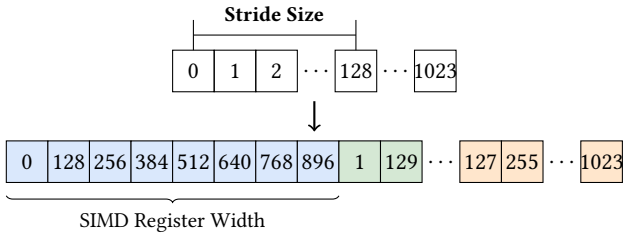


**Figure 4: Shuffled layout for SIMD decompression of 64-bit integers. We shuffle every 1024 bytes, with each AVX512 register holding eight 64-bit integers, resulting in a stride of 128 between values in each block.**

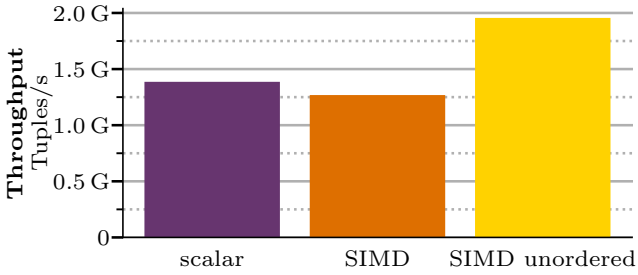


**Figure 5: Throughput of a delta-encoded column using scalar decompression, a SIMD loop that preserves order (using the scatter instruction), and a SIMD loop that does not (using a simple store instruction).**

**Writing back**. Scattering involves writing each element in a SIMD vector to its own specific memory address rather than to consecutive memory addresses. The scatter instruction is quite expensive, and when looking at the decompression execution time, most of it is spent in this instruction. As shown in Figure 5, throughput only improves when writing the data back unordered with a simple store instead of a scatter instruction. Consequently, delta decompression does not benefit from using SIMD.

## 3.4 Predicated Scans

Another reason for accessing tuples is to evaluate predicates on the data. Umbra allows direct evaluation of predicated scans directly on compressed data. However, the evaluation approach must be adjusted based on the compression technique.

For FOR compression, this involves adjusting the predicates according to the reference value. This is useful if the data does not need decompression but is just needed to construct a match vector for other attributes.

Since delta encoding has dependencies between values, the predicates can only be evaluated on the decompressed data. Therefore, the code is similar to the one used for range/matches decompression; it only contains one more check to see if the decompressed value matches the predicate. For this use case, it can also make a difference whether all deltas during compression were positive since it allows early returns. This optimization has been implemented as well.

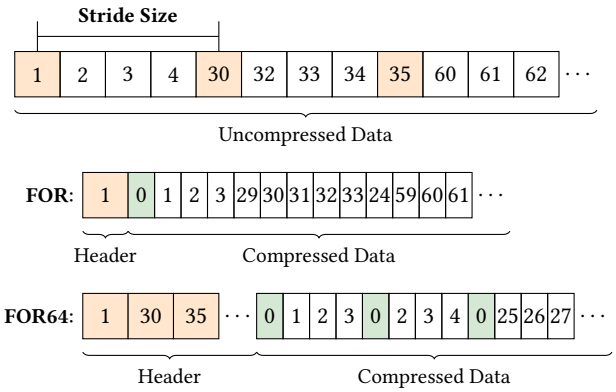


**Figure 6: Data layout for the serialized standard FOR and FOR64-encoded data.**

# 4 FOR64

Frame-of-reference (FOR) encoding processes all values in a Data Block of $2^{16}$ tuples by determining the minimum value. This minimum value is subtracted from all integers in this column, resulting in smaller integers that are stored in either 8 or, when necessary, 16 bits. For decompression, the minimum is stored in the header at the beginning of the Data Block and is added back to all values in a data block. The layout for this standard FOR approach can be seen in Figure 6.

## 4.1 Data Layout

Building on the concept of data points from delta encoding, one could extend the existing FOR encoding in a similar manner. To enhance compression, instead of storing a single minimum value per Data Block, one could store the minimum value for equally-sized sub-blocks, e.g., 64 tuples. This approach allows for random access (with an additional computation to retrieve the correct reference value) and improves compression ratios, similar to delta encoding. This is because the delta to the minimum per chunk would generally be smaller than the difference between all values and a single frame of reference for the entire data block.

We implemented this new compression scheme with the layout shown in Figure 6. Unlike the visualization, which uses a stride size of 4 for simplicity, the implementation uses a fixed stride size of 64 tuples.

## 4.2 Decompression

Decompression of FOR64-encoded data is similar to standard FOR decompression, where a reference value is added to the stored value before writing it back. In the case of FOR64, the reference has to be loaded from the header multiple times during the decompression of one Data Block. The process also differs depending on whether decompression is for a range or a match vector.

For a range, the appropriate reference is loaded from the header and the values are decompressed until the value which belongs to the next block of 64 tuples. The next tuples are then decompressed in chunks, first loading the reference value and then doing 64 additions

```
// Align the first block of 64
alignedFrom = min(r.begin + (64 - r.begin % 64), r.end)
for (i = m.begin, i != alignedFrom; ++i)
  *(writer++) = reader[i] + readerHeader[r.begin / 64]
// Decompress full blocks of 64
to64 = alignedFrom + (((to - alignedFrom) >> 6) << 6)
for (i = alignedFrom; i != to64;)
  ref = readerHeader[i / 64]
  for (j = i + 64; i != j; ++i)
    *(writer++) = reader[i] + ref
// Decompress the last block
for (i = to64; i != r.end; ++i)
  *(writer++) = reader[i] + readerHeader[to64 / 64]
```

**Listing 4: Decompressing FOR64-encoded data over a range of tuples.**

```
for (i = r.begin, i != r.end; ++i)
  ref = readerHeader[i / 64]
  *(writer++) = ref + reader[i]
```

**Listing 5: Decompressing of FOR64-encoded data given a match vector.**

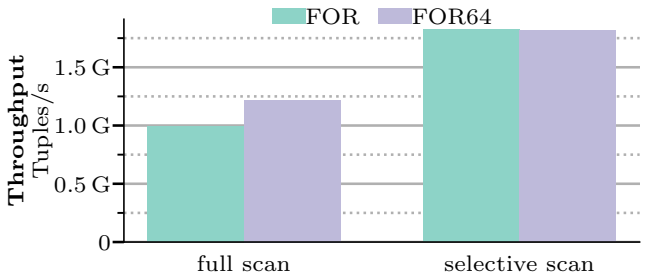


**Figure 7: Throughput of a full and selective scan on a column compressed using either the standard FOR or FOR64 implementations.**

and write-backs. The last block, which may contain less than 64 tuples, is then handled separately again, as illustrated in Listing 4.

For match vectors, decompression occurs selectively for each value whose index is in the match vector. Therefore, we load the data point from the header belonging to the index and add the stored delta. The exact implementation can be seen in Listing 5.

## 4.3 Comparison with standard FOR

We evaluated our approach by analyzing the compression factors and decompression speeds of the `ps_partkey` column from the `partsupp` table. The standard FOR implementation only compresses to 16-bit integers, whereas FOR64 compresses to 8-bit integers. As Figure 7 shows, range decompression over the entire column is 23% faster with FOR64 compared to the standard FOR implementation. Both implementations achieve similar speeds in the case of a scan with 10% selectivity.

# 5 EVALUATION

We evaluated our implementations by comparing them to operations on uncompressed data or on data compressed using different

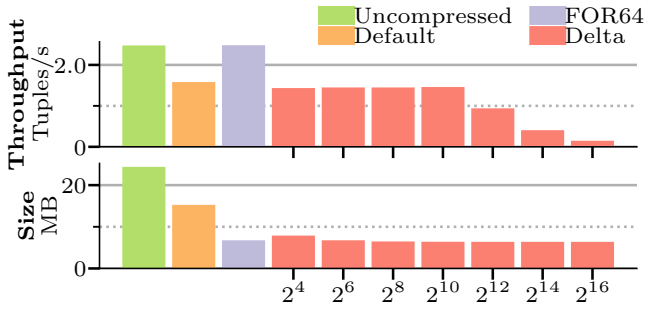


**Figure 8: Throughput of a full scan over `l_orderkey` for uncompressed data and data compressed with various schemes. For delta encoding, different stride sizes were tested.**

schemes. We only look at the decompression speeds, since compression should only happen once or rarely, while decompression might happen any time analytics are performed on the data. All benchmarks were performed on an AMD Ryzen 9 7900X 12-Core Processor and are run single-threaded in our database system Umbra. The benchmarks involve performing a `sum` operation over the compressed column to ensure the data is decompressed.

The benchmarks were conducted on the same column, which was compressed using different methods:

*Uncompressed*: The colum is stored in uncompressed form.

*Default*: Umbra applies its compression logic to determine a possible compression scheme, such as FOR and Dictionary encoding.

*FOR64*: The column is compressed using our new FOR64 approach.

*Delta*: The column is compressed using delta encoding.

For the delta encoding scheme, we additionally conduct experiments on the effect of different stride sizes.

**Full scan performance**. We first looked at the performance of our implementations for a full scan over the column `l_orderkey` of the `lineitem` table. This column is compressed to 1-byte values using both delta encoding and FOR64 encoding. Without these additional compression schemes, the standard Umbra compression uses FOR with a truncation to 2 bytes and, for some Data Blocks, dictionary encoding.

For the delta encoding, the stride size parameter is crucial. A stride size that is too small leads to a lower compression ratio, while a stride size that is too large hinders parallel execution due to increased workload per thread. Since at a stride size of $\leq 4$ the FOR encoding with 2-byte truncation occupies the same amount of storage as a 1-byte delta encoding, we began with a stride size of 16 and then increased it by multiples of 4 until reaching the Data Block size of $2^{16}$ tuples. The results for a full unselective scan and the storage sizes can be seen in Figure 8.

For a non-predicated scan, there is no performance difference between data with a stride size of 1024 or less. This is because the morsel size is also 1024 tuples, and the scan does not benefit from intermediate values being stored within a range of a thread. Conversely, a stride size larger than 1024 leads to a drop in performance since multiple threads redundantly decompress their morsels and the dependencies are not sufficiently broken up.

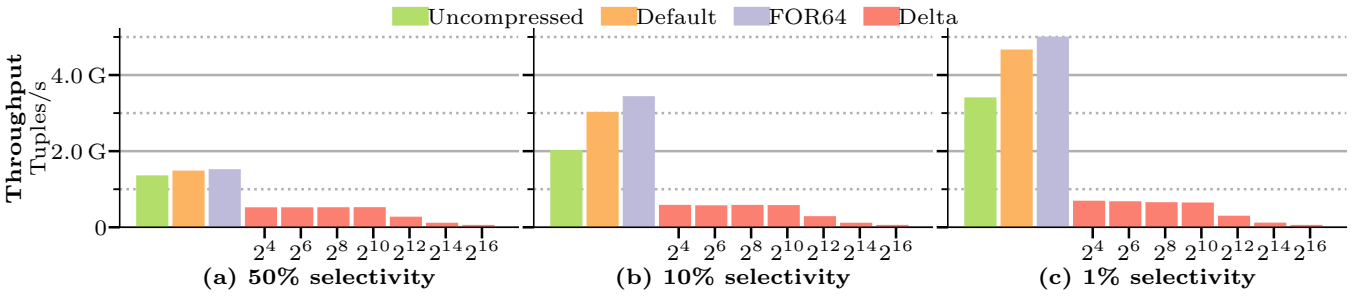


**Figure 9: Throughput of scans with different selectivities over `l_orderkey` for uncompressed data and data compressed with various schemes. For delta encoding, different stride sizes were tested.**

Delta and FOR64 encoding achieve the best compression factor, compressing by approximately a factor 4, which is twice as much as before. Performance-wise, scans on the FOR and dictionary-encoded data are as fast as the ones on delta-encoded data. Both the uncompressed scan and the scan over FOR64-encoded data achieve higher throughput due to SIMD vectorization. For the uncompressed column, a vectorized `memcpy` is used. The main decompression loop for FOR64 is simple enough that the compiler can generate vectorized code for it. Since the decompression of the delta-encoded data is more complex, the generated code remains scalar.

**Decompression with Match Vectors**. We also looked at selective queries, where the predicate is on a different column than the delta-encoded column. When executing selective queries that use match vectors when decompressing, delta decompression performs worse than all other methods, as shown in Figure 9.

This is due to the dependencies inside the delta-encoded column, which turn a match vector decompression into a range decompression. While other schemes allow random access, where only the necessary indices are accessed, delta compression requires decompressing the entire range from the lowest to the highest index. This means there is little difference in performance between a scan of higher or lower selectivity. Since a thread also stays in its 1024 tuple bound in the case of a match vector, i.e., the first and last index in the match vector are never more than 1024 tuples apart, there is no way of circumventing this for stride sizes $\geq 1024$.

In contrast, FOR64 encoding does not suffer from this limitation. It performs comparably to working with uncompressed or FOR/dictionary-encoded data. FOR64 is 12–47% faster than the uncompressed reference and 3–7% faster than the FOR/dictionary reference.

**Optimizing Match Vector extraction for Delta Compression**. If the stride size is smaller than the morsel size, the match vector extraction can be optimized by skipping ranges of values whenever possible instead of decompressing each value sequentially. Specifically, if the following match vector index is farther away than the next data point in the header, these values are skipped. This optimization can improve performance, as shown in Figure 10, but its benefits are most noticeable with high selectivity and a small stride size. Even with this optimization, performance remains 2.6 times worse than on a column using the FOR64 encoding.

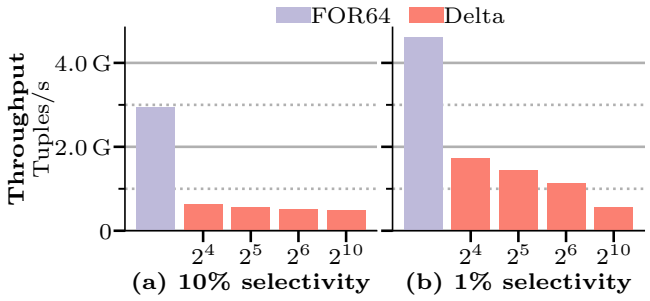


**Figure 10: Throughput of scans with 10% and 1% selectivity over `l_orderkey` for delta and FOR64-encoded data. For delta encoding, stride sizes $\leq 2^{10}$ use an optimization to avoid unnecessary decompression.**

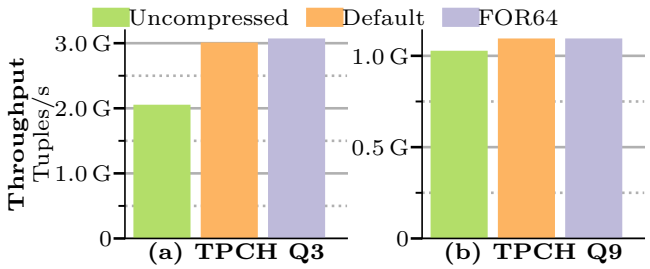


**Figure 11: Throughput of TPC-H Q3 and Q9 using different compression techniques.**

**Full Query Performance**. Lastly we looked at full query performance with TPC-H queries 3 and 9, which both work on the compressed `o_orderkey` and `l_orderkey` columns. We compare the FOR64 implementation with the default, focusing solely on these two differently compressed columns. Additionally, we also compare these results to uncompressed data, where all data involved is stored uncompressed. These benchmarks were done multithreaded with 12 threads and the results can be seen in Figure 11.

The results are consistent with previous findings: compression can enhance throughput, and the new FOR64 implementation performs comparably or even slightly better than the old FOR implementation or dictionary encoding.

## 6 FUTURE WORK & CONCLUSION

We will now look at some possible improvements to our compression schemes and summarize our findings.

**Runaway values**. Both delta and FOR64 encoding enhance the compression of columns that would otherwise be stored uncompressed or with a worse compression ratio. However, even with these new compression schemes, some columns, such as in the JOB dataset, are still stored uncompressed. Often, the majority of the deltas are very small because the data is almost sorted, but occasionally, an unsorted value appears or a new range of sorted values begin. For instance, in the `movie_id` column in `movie_info`, there are 120382 deltas that do not fit in a signed 16-bit integer and 337194 deltas that would not fit in a signed 8-bit integer, representing approximately 0.8% or 2.2% of all deltas. To compress these columns, we could

(1) store outlier values with no connection to those values around them (i.e., large deltas to their respective proceeding and succeeding values) separately in larger integer types and use a bitmap to identify their location, similar to NULL values.
(2) choose different byte-widths for each 64-tuple chunk in the FOR64 encoding, so only specific chunks are stored in 4 or 8-byte integers instead of an entire Data Block. This approach would require an additional runtime lookup to load the values correctly.

**Conclusion**. In this work, we implemented two new integer compression schemes, delta encoding and FOR64 encoding, into the Umbra database system. Both schemes perform better on certain integer data types than FOR and dictionary encoding. We can achieve compression ratios up to 4 when compressing 4-byte uncompressed data to 1-byte truncated data. While both schemes offer similar compression ratios, delta compression suffers from dependencies between values, making random access difficult to implement efficiently. In contrast, the new FOR64 encoding avoids this problem, offering slightly better performance than standard FOR encoding and achieving much better compression. While the full scan performance is comparable between delta and FOR64 encoding, delta encoding is 2.3-4.2× slower in highly selective scans. Therefore, we recommend using a smarter FOR encoding approach instead of delta encoding, as it achieves the same compression but is easier to vectorize and supports random access.

## REFERENCES


[1] Parquet. https://parquet.apache.org/docs/file-format/data-pages/encodings/. Accessed: 2024-03-29.
[2] SAP. https://help.sap.com/docs/HANA_SERVICE_CF/6a504812672d48ba865f4f4b268a881e/bd9017c8bb571014ae79efaeb46940f3.html. Accessed: 2024-03-29.
[3] D. Abadi, P. A. Boncz, S. Harizopoulos, S. Idreos, and S. Madden. The design and implementation of modern column-oriented database systems. *Found. Trends Databases*, 5(3):197–280, 2013.
[4] A. Afroozeh and P. A. Boncz. The fastlanes compression layout: Decoding >100 billion integers per second with scalar code. *Proc. VLDB Endow.*, 16(9):2132–2144, 2023.
[5] C. Anneser, L. Vogel, F. Gruber, M. Bandle, and J. Giceva. Programming fully disaggregated systems. In *HotOS*, pages 188–195. ACM, 2023.
[6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, 2008.
[7] B. Dageville, T. Cruanes, M. Zukowski, V. Antonov, A. Avanes, J. Bock, J. Claybaugh, D. Engovatov, M. Hentschel, J. Huang, A. W. Lee, A. Motivala, A. Q. Munir, S. Pelley, P. Povinec, G. Rahn, S. Triantafyllis, and P. Unterbrunner. The snowflake elastic data warehouse. In *SIGMOD Conference*, pages 215–226. ACM, 2016.
[8] M. Fehér, D. E. Lucani, and I. Chatzigeorgiou. An adaptive column compression family for self-driving databases. In *ADMS@VLDB*, pages 47–57, 2022.
[9] L. Heinzl, B. Hurdelhey, M. Boissier, M. Perscheid, and H. Plattner. Evaluating lightweight integer compression algorithms in column-oriented in-memory DBMS. In *ADMS@VLDB*, pages 26–36, 2021.
[10] M. Kuschewski, D. Sauerwein, A. Alhomssi, and V. Leis. Btrblocks: Efficient columnar compression for data lakes. *Proc. ACM Manag. Data*, 1(2):118:1–118:26, 2023.
[11] H. Lang, T. Mühlbauer, F. Funke, P. A. Boncz, T. Neumann, and A. Kemper. Data blocks: Hybrid OLTP and OLAP on compressed storage using both vectorization and compilation. In *SIGMOD Conference*, pages 311–326. ACM, 2016.
[12] V. Leis, P. A. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: a numa-aware query evaluation framework for the many-core age. In *SIGMOD Conference*, pages 743–754. ACM, 2014.
[13] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9):539–550, 2011.
[14] T. Neumann and M. J. Freitag. Umbra: A disk-based system with in-memory performance. In *CIDR*. www.cidrdb.org, 2020.
[15] O. Polychroniou and K. A. Ross. Towards practical vectorized analytical query engines. In *DaMoN*, pages 10:1–10:7. ACM, 2019.
[16] M. Pöss and D. Potapov. Data compression in oracle. In *VLDB*, pages 937–947. Morgan Kaufmann, 2003.
[17] L. Poutievski, O. Mashayekhi, J. Ong, A. Singh, M. M. B. Tariq, R. Wang, J. Zhang, V. Beauregard, P. Conner, S. D. Gribble, R. Kapoor, S. Kratzer, N. Li, H. Liu, K. Nagaraj, J. Ornstein, S. Sawhney, R. Urata, L. Vicisano, K. Yasumura, S. Zhang, J. Zhou, and A. Vahdat. Jupiter evolving: transforming google's datacenter network via optical circuit switches and software-defined networking. In *SIGCOMM*, pages 66–85. ACM, 2022.
[18] M. Raasveldt and H. Mühleisen. Data management for data science - towards embedded analytics. In *CIDR*. www.cidrdb.org, 2020.
[19] M. Zukowski, S. Héman, N. Nes, and P. A. Boncz. Super-scalar RAM-CPU cache compression. In *ICDE*, page 59. IEEE Computer Society, 2006.