Till Kolditz, Dirk Habich, Dmitrii Kuvaiskii, Wolfgang Lehner, Christof Fetzer

Needles in the Haystack - Tackling Bit Flips in Lightweight Compressed

Erstveröffentlichung in / First published in:

Data Management Technologies and Applications: 4th International Conference. Colmar, 20.-22.07.2015. Springer, S. 135-153. ISBN 978-3-319-30162-4.

DOI: http://dx.doi.org/10.1007/978-3-319-30162-4_9

Final edited form was published in "Data Management Technologies and Applications 4th International Conference. Colmar 2015", S. 135-153. ISBN: 978-3-319-30162-4
https://doi.org/10.1007/978-3-319-30162-4_9

Needles in the Haystack — Tackling Bit Flips in Lightweight Compressed Data

Till Kolditz[1], Dirk Habich[1(✉)], Dmitrii Kuvaiskii[2], Wolfgang Lehner[1], and Christof Fetzer[2]

[1] Technische Universität Dresden, Database Systems Group, 01062 Dresden, Germany
{Till.Kolditz,Dirk.Habich,Wolfgang.Lehner}@tu-dresden.de
[2] Technische Universität Dresden, Systems Engineering Group, 01062 Dresden, Germany
{Dmitrii.Kuvaiskii,Christof.Fetzer}@tu-dresden.de

Abstract. Modern database systems are very often in the position to store their entire data in main memory. Aside from increased main emory capacities, a further driver for in-memory database system has been the shift to a column-oriented storage format in combination with lightweight data compression techniques. Using both mentioned software concepts, large datasets can be held and efficiently processed in main memory with a low memory footprint. Unfortunately, hardware becomes more and more vulnerable to random faults, so that e.g., the probability rate for bit flips in main memory increases, and this rate is likely to escalate in future dynamic random-access memory (DRAM) modules. Since the data is highly compressed by the lightweight compression algorithms, multi bit flips will have an extreme impact on the reliability of database systems. To tackle this reliability issue, we introduce our research on error resilient lightweight data compression algorithms in this paper. Of course, our software approach lacks the efficiency of hardware realization, but its flexibility and adaptability will play a more important role regarding differing error rates, e.g. due to hardware aging effects and aggressive processor voltage and frequency scaling. Arithmetic AN encoding is one family of codes which is an interesting candidate for effective software-based error detection. We present results of our research showing tradeoffs between compressibility and resiliency characteristics of data. We show that particular choices of the AN-code parameter lead to a moderate loss of performance. We provide evaluation for two proposed techniques, namely AN-encoded Null Suppression and AN-encoded Run Length Encoding.

1 Introduction

Data management is a core service for every business or scientific application in today's data-driven world. The data life cycle comprises different phases starting from understanding external data sources and integrating data into a common database schema. The life cycle continues with an exploitation phase

Final edited form was published in "Data Management Technologies and Applications 4th International Conference. Colmar 2015", S. 135-153. ISBN: 978-3-319-30162-4
https://doi.org/10.1007/978-3-319-30162-4_9

by answering queries against a potentially very large database and closes with archiving activities to store data with respect to legal requirements and cost efficiency. While understanding the data and creating a common database schema is a challenging task from a modeling perspective, efficiently and flexibly storing and processing large datasets is the core requirement from a system architectural perspective [17,28].

With an ever increasing amount of data in almost all application domains, the storage requirements for database systems grows quickly. In the same way, the pressure to achieve the required processing performance increases, too. To tackle both aspects in a consistent uniform way, data compression as software concepts plays an important role. On the one hand, data compression drastically reduces storage requirements. On the other hand, compression also is the cornerstone of an efficient processing capability by enabling "in-memory" technologies. As shown in different papers, the performance gain of in-memory data processing for database systems is massive because the operations benefit from its higher bandwidth and lower latency [1,6,14,18].

Aside from the developments in the data compression domain, the hardware sector has seen important developments, too. Servers with terabytes of main memory are available for a reasonable price, so that the entire data pool in a compressed form can be kept and processed completely in main memory. In order to increase main memory density and to put more functionality into integrated circuits (ICs), transistor feature sizes are decreased more and more. On the one hand, this leads to performance improvements in each hardware generation. On the other hand, ICs become more and more vulnerable to external influences like cosmic rays, electromagnetic radiation, low voltages, and heat dissipation. Data Centers already face crucial error rates in dynamic random-access memory (DRAM) [13,25] including multi bit flips which cannot be handled by typical SECDED[1] ECC DRAM anymore. These error rates are likely to increase in the future, and will become a major challenge for in-memory database systems. We argue that ECC DRAM alone is not the silver bullet, because the employed codes are statically integrated into hardware with fixed parameters.

Generally, the field of error correcting codes as well as the field of data compression techniques are well-established. In order to tackle the above mentioned resiliency challenge for in-memory database system, we propose to tightly combine existing techniques from the corresponding files in an appropriate way: *resiliency-aware data compression techniques*. Of course, our software approach lacks the efficiency of resiliency-aware hardware realization like ECC DRAM, but its flexibility and adaptability will play a more important role regarding differing error rates, e.g., due to hardware aging effects and aggressive processor voltage and frequency scaling. Our main idea is to minimize the amount of useful information (bits)—using data compression—which are then enriched by redundant information (bits) to protect against bit flips. As a side constraint, our resiliency-aware compressed data should allow to directly work on that data representation without explicitly decompressing and re-encoding the data. This

[1] Single-error correcting and double-error detecting.

Final edited form was published in "Data Management Technologies and Applications 4th International Conference. Colmar 2015", S. 135-153. ISBN: 978-3-319-30162-4
https://doi.org/10.1007/978-3-319-30162-4_9

constraint is important to achieve query processing performance expected by in-memory database systems. Furthermore, error detection should be possible in an online fashion, so that wrong results can be excluded to a well-defined degree.

Our Contribution. To show the potential and challenges in this research direction, we present our first research results in this paper. From the field of error correcting codes, we have chosen the family of arithmetic AN codes as a very promising alternative (or complementary) to ECC DRAM, since its very nature allows to do arithmetic operations – including comparisons – without the need of decoding. Consequently, arithmetic AN codes are suitable for both transactional and analytical database workloads. From the data compression domain, we decided to use two heavily-used lightweight techniques: Null Suppression [1,22], and Run Length Compression [1]. In detail, our contributions are as follows:

1. We show how to tightly combine arithmetic AN encoding with Null Suppression [1,22], and Run Length Compression [1] as concrete examples for our *resiliency-aware data compression techniques* or *AN-encoded lightweight data compression techniques*. As we are going to show, the combination approach differs and depends on various factors.
2. We introduce our *AN-encoded data compression techniques* in our data compression modularization concept. Generally, our modularization concept offers an efficient and an easy-to-use way to describe, to compare, and to adapt (AN-encoded) lightweight data compression techniques.
3. We provide an analysis of how the basic parameter A of AN encoding can be chosen to detect various amounts of bit flips.
4. We show that there are "good" As with very low performance penalties to enable online error detection for compressed data.
5. We provide a performance evaluation for the "good" As for both AN-encoded Null Suppression and AN-encoded Run Length Compression.

Outline. The remainder of this paper is structured as follows: In Sect. 2, we present related work with a brief overview of existing lightweight compression techniques and give a detailed insight into AN encoding in Sect. 3. Then, we present how to integrate AN encoding with two compression schemes in Sect. 4. Next, we present our evaluation in Sect. 5, where we discuss "good" As and provide throughput comparisons for one of the "good" As. Finally, we conclude the paper in Sect. 6.

2 Related Work

Before we present our novel approach of resiliency-aware data compression techniques, we briefly review related work on (1) lightweight data compression techniques frequently used in-memory database system in Sect. 2.1 and (2) generic and database-specific resilience mechanism in Sect. 2.2 and 2.3.

Final edited form was published in "Data Management Technologies and Applications 4th International Conference. Colmar 2015", S. 135-153. ISBN: 978-3-319-30162-4
https://doi.org/10.1007/978-3-319-30162-4_9

2.1 Lightweight Data Compression Techniques

In the area of conventional data compression a multitude of approaches exists. Classic compression techniques like arithmetic coding [31], Huffman [12], or Lempel-Ziv [32] achieve high compression rates, but the computational effort is high. Therefore, those techniques are usually denoted as heavyweight. Especially for in-memory database systems, a variety of lightweight compression algorithms has been developed. These achieve good compression rates similar to heavyweight methods by utilizing context knowledge, but they require much faster compression and decompression.

The main archetypes or classes of lightweight compression techniques are dictionary compression (DICT) [2,5,16], delta coding (DELTA) [18,22], frame-of-reference (FOR) [7,33], Null Suppression (NS) [1,21,22,27], and Run-Length Encoding (RLE) [3,22]. DICT replaces each value by its unique key. DELTA and FOR represent each value as the difference to its predecessor or a certain reference value, respectively. These three well-known techniques try to represent the original data as a sequence of small integers, which is then suited for actual compression using a scheme from the family of NS. NS is the most well-studied kind of lightweight compression. Its basic idea is the omission of leading zeros in small integers. Finally, RLE tackles uninterrupted sequences of occurrences of the same value, so-called runs. In its compressed format, each run is represented by its value and length, i.e., by two uncompressed integers. Therefore, the compressed data is a sequence of such pairs.

2.2 Generic Resilience Mechanisms

Increased memory density, decreased transistor feature sizes and more are major drivers in the area of hardware development. On the one hand, this leads to performance improvements in each hardware generation. On the other hand, the hardware becomes more and more vulnerable to external influences. As several researches have already stated, especially main memory becomes a severe cause for hardware based failures [13,15,19,25,26]. These errors can be classified into *static or hard errors* as permanently corrupted bits and *dynamic or soft errors* as transiently corrupted bits. In particular, dynamic errors are produced, e.g., by cosmic rays, electromagnetic radiation, low voltage and increased heat dissipation.

While dynamic error rates are still quite low, it is predicted that they increase substantially in the near future [13]. Moreover, dynamic errors already have a significant impact on large-scale applications on massive data sets. The field of fault tolerance against dynamic memory errors is not new and several techniques are well-known. A generally applicable approach is executing the same computation multiple times. In this case, any dynamic error can be detected by comparing the final results – except when both results have the very same error which is just assumed to not happen. The most well-known technique in this class is Triple Modular Redundancy. Error detection and error correction codes represent a second class. In this case, the coding schemes introduce redundancy

Final edited form was published in "Data Management Technologies and Applications 4th International Conference. Colmar 2015", S. 135-153. ISBN: 978-3-319-30162-4
https://doi.org/10.1007/978-3-319-30162-4_9

to the data [20]. Regarding DRAM bit flips, the most commonly used approach is hardware-based (72,64)-Hamming ECC [20], which realizes single-error correction and double-error detection (SECDED). Many other general coding algorithms are available, whereas the enhanced coding schemes are more robust, however their coding results in higher memory overhead and higher computational costs. Generally, the major problem of ensuring a low dynamic error probability by employing generally applicable techniques is dramatically increasing costs for memory and computational power.

2.3 Database-Specific Resilience Mechanism

In the past, several techniques have been presented to deal with certain error classes. To our best knowledge, no research was done in the field of databases to protect in-memory data against arbitrary bit flips, except our own investigations on error detecting B-Trees [15]. We presented software based adaptations for B-Trees, a widely used database index structure, to cope with increasing bit flip rates in main memory. We showed that pointer sanity checks, parity bits and checksums can deliver comparable or better error detection on commodity hardware compared to ECC hardware, since they are able to detect more than 2 bit flips in 8-byte words. Furthermore, we showed that checksums are able to detect more bit flips and provide higher reliability which is highly desired for database systems.

In the field of databases, other relevant related work mainly concentrates on handling errors during I/O operations, or regards situations where the system inadvertently writes to wrong memory regions, e.g. due to software bugs like buffer overflows or broken pointers. For instance, [8,9] harden also the well-known B-Tree and variants against errors during I/O operations or against certain other tree corruptions. On the one hand these techniques are offline methods, which means they are periodically executed. On the other hand, they may be very heavy-weight, especially when comparing entries between several indices, and may not be suited for online error detection. Furthermore, bit flips may lead to false positives and false negatives when querying such trees between these maintenance checks.

Sullivan et al. [29] deal with corruptions due to arbitrary writes by employing hardware memory protection for individual pages. Memory pages are protected using hardware directives and the protection is removed only when accessing the pages through a special interface. The routines for protecting and unprotecting require kernel calls which leads to high performance penalties. Additionally, while a page is unprotected other threads may still corrupt data. Furthermore, this does not help against bit flips as they are not induced by stray writes, but by the hardware itself. Furthermore, Bohannon et al. [4] handle the case for in-memory database systems by computing XOR-checksums over certain protected memory regions. A codeword table is maintained which stores the original checksums. Pages are then later validated against this table. This again helps detecting undesired, arbitrary writes, but bit flips may corrupt the codeword table and, e.g., correct pages may then mistakenly be regarded as corrupted.

Final edited form was published in "Data Management Technologies and Applications 4th International Conference. Colmar 2015", S. 135-153. ISBN: 978-3-319-30162-4
https://doi.org/10.1007/978-3-319-30162-4_9

3 Error Detection by Arithmetic Codes

To tackle our vision of *resiliency-aware data compression techniques*, we decided to utilize arithmetic codes[2] as our resilience technique in a first step. Arithmetic codes are a long known technique to detect hardware errors at runtime caused by transient (e.g., dynamic bit flips) and permanent (e.g., stuck-at-1) hardware faults [23]. This is achieved by adding redundancy to processed data, i.e., a larger domain of possible data words is created. The domain of possible words contains the smaller subset of valid code words – the so-called encoded data items. Arithmetic codes are preserved by correct arithmetic operations, that is, a correctly executed operation taking valid code words as input produces a result that is also a valid code word.

3.1 Basic Idea of an Encoding

The underlying idea of AN encoding is simple: multiply each data word n by a predefined constant A, i.e., the code word $\hat{n}$ is computed as:

$$\hat{n} = n \cdot A \tag{1}$$

As a result of this multiplication (*encoding*), the domain of values expands such that only the multiples of A become valid code words, and all other values are considered non-code. As an example, if one wants to encode a set of 2-bit numbers $\{0, 1, 2, 3\}$ with $A = 11$, then the set of code words is $\{0, 11, 22, 33\}$, while $1, 10, 34$ are all examples of non-code words.

If a bit flip affects an encoded value, the corrupted value becomes non-code with a probability of $(A-1)/A$. If $\hat{n} = 11$ and the least significant bit is flipped, then the new value $\hat{n}_{er} = 10$ and is non-code. To detect this fault, we have to check if the value is still a multiple of A:

$$\hat{n} \mod A = 0 \tag{2}$$

Finally, to decode the value, we have to divide the code word $\hat{n}$ by A:

$$n = \hat{n}/A \tag{3}$$

For any given native data width X – usually $X \in \{8, 16, 32, 64\}$ – processors' integer arithmetic modular arithmetic, i.e. the equation $a * b = c$ implicitly transforms into $|a*b| \equiv |c| \mod 2^X$ for unsigned integers, or $a*b \equiv c \mod 2^{X-1}$ for signed integers. By that, for several A's there exists a multiplicative inverse A^{-1} so that

$$n = \hat{n}/A = \hat{n} * A^{-1} \tag{4}$$

For instance, $641^{-1} \equiv 6700417 \mod 2^{32}$ and Table 1 lists the available inverses for 32-bit unsigned integers for the given A's – in our case any even number has no inverse. Consequently, for some A's the division is replaced by a multiplication which is usually much faster on modern processors.

[2] Please note that some codes for lossless data compression are also called arithmetic codes. These are not equivalent with the ones used throughout this paper.

Final edited form was published in "Data Management Technologies and Applications 4th International Conference. Colmar 2015", S. 135-153. ISBN: 978-3-319-30162-4
https://doi.org/10.1007/978-3-319-30162-4_9

3.2 Beneficial Features of an Encoding

Generally, arithmetic code or AN-encoding offers some features that are beneficial for database systems. One of the features of AN encoding is the ability to directly process encoded data, i.e., there is no need to decode values before working on them. Most database-related operations can be performed on encoded values; these operations include addition, subtraction, negation, comparisons, etc. For example, the addition of two valid code words $11 + 22 = 33$ produces an expected code word, and 11 is less than 22 just like their original counterparts. Encoded multiplication and division are also possible, but require some adjustments.

This *encoded processing* feature is beneficial for in-memory database systems. AN-encoded data words can be read from main memory, processed using complex queries and stored back without the need for intermediate decoding, which reduces the overhead for resiliency mechanism. Examples of database operations on encoded data include scans, projections, aggregate computations, joins, etc.

3.3 Application Challenge

AN encoding is an arithmetic encoding scheme, allowing certain arithmetic operations directly on encoded data with relatively little overhead as well as multiplication and division with higher overhead. However, AN encoding does not pose any restrictions on a value of A. This constant must be carefully chosen to suit the needs of a particular application. The choice of A affects three parameters: *fault coverage*, *memory footprint*, and *encoding/decoding performance*. As a rule of thumb, greater values of A result in higher fault coverage, higher memory footprint and worse performance. The challenge is to find an A providing sufficiently high fault detection rate at a low cost of memory blow-up and performance slowdown.

In general, some "good" A's with the best trade-offs can be found. In terms of fault coverage, there is no known formula to find the best A, so the researchers resort to experimental results [11]. Memory blow-up depends on the size of A in bits; for example, encoding one 22-bit integer with a 10-bit A requires $22 + 10 = 32$ bits, i.e., an increase of 45%. Finally, performance slowdown can be negligible during encoding (since multiplication requires only $2 - 3$ CPU cycles, see the note on multiplicative inverses above), but can be a bottleneck during checks and decoding (since division is an expensive CPU instruction). To alleviate this decoding impact, A must be chosen such that the division operation is substituted by a sequence of shifts, adds, and multiplies [30].

4 Resiliency-Aware Data Compression

To the best of our knowledge, nowadays no additional information is added to explicitly detect bit flip corruption of compressed data in main memory database systems. In order to tackle an increasing bit flip error rate, in particular

Final edited form was published in "Data Management Technologies and Applications 4th International Conference. Colmar 2015", S. 135-153. ISBN: 978-3-319-30162-4
https://doi.org/10.1007/978-3-319-30162-4_9

for dynamic errors, we want to tightly combine techniques from both fields of lightweight data compression and resilience techniques like AN encoding. On the one hand, lightweight data compression reduces or eliminates data redundancy to represent data using less bits. On the other hand, resilience techniques introduces data redundancy to detect bit flips. Therefore, both fields have opposing aims and combined approaches have to be carefully designed, so that benefits of both fields remain. Anyways, by compressing data, (almost) exclusively those bits which contain actual information are taken into account by the AN-encoding process. As we are going to show later, based on a well-defined and specific approach, the overhead of resiliency-aware data compression is less compared to uncompressed data, so that the approach is beneficial for database systems.

As next, we are going to present two specific AN-encoded compression scheme extensions: (i) AN-encoded Null Suppression in Sect. 4.1 and (ii) AN-encoded Run-Length Compression in Sect. 4.2.

4.1 AN-encoded Null Suppression

Null Suppression (NS) is the most well-studied kind of lightweight data compression technique. Its basic idea to the omission of leading zeros in small integers. This technique further distinguishes between bit-wise and byte-wise null suppression where either all leading zero bits or leading zero bytes containing only zero bits are stripped off. Usually, some kind of compression mask denotes how many bits or bytes were omitted from the original value. Decompression works by adding the leading zeros back.

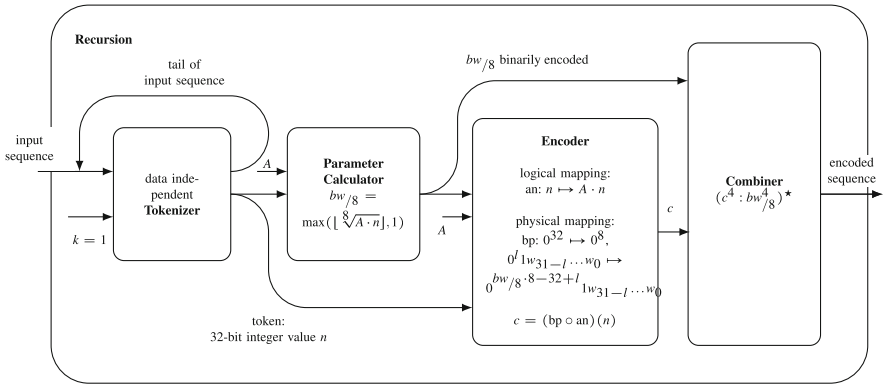


Fig. 1. Compression Scheme for Modularization for AN-encoded Null Suppression.

General Idea

The general idea of our AN-encoded Null Suppression technique is illustrated in Fig. 1. The illustration is based on our modularization concept for lightweight

Final edited form was published in "Data Management Technologies and Applications 4th International Conference. Colmar 2015", S. 135-153. ISBN: 978-3-319-30162-4
https://doi.org/10.1007/978-3-319-30162-4_9

data compression methods [10]. Our scheme is a recursion module for subdividing data sequences several times. The first module in each recursion is a `Tokenizer` splitting the input sequence in finite subsequences or single values at the finest level of granularity. For that, the `Tokenizer` can be parameterized with a calculation rule. The finite output sequence of the `Tokenizer` serves as input for the `Parameter Calculator`, which is our second module. Parameters are often required for the encoding and decoding. Therefore, we introduce this module, whereas this module knows special rules (parameter definitions) for the calculation of several parameters. Our third module as depicted in Fig. 1 is the `Encoder`, which can be parameterized with a calculation rule for the processing of an atomic input value, whereas the output of the `Parameter Calculator` is an additional input. Its input is a token that cannot or shall not be subdivided anymore. In practice the `Encoder` gets a single integer value to be mapped into a binary code. The fourth and last module is the `Combiner`. It determines how to arrange the output of `Encoder` together with the output of the `Parameter Calculator`. Generally, these four main modules including the illustrated assembly in Fig. 1 are enough to specify a large number of lightweight data compression algorithms.

Our AN-encoded Null Suppression algorithm works as follows and is depicted in Fig. 1: We use a very simple `Tokenizer` outputting single integer values of an input data sequence. This `Tokenizer` instance can be characterized as *data independent* and *non-adaptive*, whereas only the beginning of the data sequence has to be known. For each value, the `Parameter Calculator` determines the number of necessary bytes (omission of leading zero bytes), whereas each value is multiplied by an value A for resiliency before. The corresponding formula is depicted in Fig. 1. The determined number of bytes is used in the subsequent `Encoder` to compute the bit representation of the AN-encoded value. The binary representations for whole compressed AN-encoded values are concatenated in the `Combiner`, symbolized by a star. That means, our AN-encoded Null Suppression technique encodes the values first and compresses afterwards. The compression mask itself is also AN-encoded.

SIMD-Based Implementation

In recent years, research in the field of lightweight data compression has mainly focussed on the efficient implementation of the techniques on modern hardware e.g., using vectorization capabilities of modern CPUs (SSE or AVX extensions). Schlegel et al. [24] presented 4-Wise Null Suppression as vectorized version. 4-Wise NS eliminates leading zeros at byte level and processes blocks of four integer values at a time. During compression the number of leading zero bytes of each of the four values is determined. This yields four 2-bit descriptors, which are combined to an 8-bit compression mask. The compression of the values is done by a SIMD byte permutation bringing the required lower bytes of the values together. This requires a permutation mask, which is looked up in an offline-created table using the compression mask as a key. After the permutation, the code words have a horizontal layout, i.e. code words of subsequent

Final edited form was published in "Data Management Technologies and Applications 4th International Conference. Colmar 2015", S. 135-153. ISBN: 978-3-319-30162-4
https://doi.org/10.1007/978-3-319-30162-4_9

basicstyle

```
1  compress (in elements[], out buffer[])
2  {
3    for (i = 0; i < |elements|; i = i + 4)
4    {
5      n1 = elements[i] * A;
6      ...
7      n4 = elements[i+3] * A;
8      z1 = count_zero_bytes(n1);
9      ...
10     z4 = count_zero_bytes(n4);
11     mask = (z4 << 6) | (z3 << 4) | (z2 << 2) | z1;
12     buffer ← (mask * A);
13     buffer ← n1;
14     ...
15     buffer ← n4;
16   }
17 }
```

Listing 1.1. Pseudo code for AN encoded 4-wise Null Suppression. `elements` is the input array while `buffer` is the output array. `|elements|` denotes the array's number of elements.

values are stored in subsequent memory locations. The compressed data is thus a sequence of compressed blocks. The decompression simply reads the compression mask, looks up the appropriate permutation mask which reinserts the leading zeros bytes and applies the permutation. Based on that principle, we are able to introduce our resiliency-aware extension of 4-Wise NS as an efficient vectorized implementation of our AN-encoded Null Suppression technique as illustrated in Fig. 1.

Encoding and Compression. Encoded compression for NS works as follows. Listing 1.1 shows the pseudo code for a 4-Wise encoded NS scheme (processing four 32-bit integers at once in a vectorized version). There are input and output arrays to function `compress`, where `elements` stores original data and `buffer` receives the compressed and encoded data. Four data items are processed in each loop iteration (line 3). First, each item is multiplied by A (lines 5–7) and afterwards the leading zero bytes are counted (lines 8–10). This can be done by counting the leading zero bits using compiler intrinsics (`_builtin_clz()` for g++) and then dividing by 8. The bit compression mask contains the number of leading zeros. It is computed by ORing the lower 2 bits of the zero byte counts together (line 11). Finally, the mask is encoded and the compressed encoded words are stored in the output buffer (lines 12–15). Assuming a little endian system, the leading zero bytes of a compressed value are inherently overwritten by the next appended data item, by advancing the write pointer by the number of non-zero bytes of the item just written.

Final edited form was published in "Data Management Technologies and Applications 4th International Conference. Colmar 2015", S. 135-153. ISBN: 978-3-319-30162-4
https://doi.org/10.1007/978-3-319-30162-4_9

```
1  decompress (in buffer[], out elements[])
2  {
3    for (i = 0; i < |buffer|; )
4    {
5      mask = buffer[i] * A^-1;
6      if (mask % A != 0) error();
7      mask = mask * A^-1;
8      i = i + 1;
9      non_zero_bytes = mask & 0x3;
10     item = buffer[i] & (0xFFFFFFFF >> (non_zero_bytes * 8));
11     if (item % A != 0) error();
12     elements ← item * A^-1;
13     i = i + 4 - non_zero_bytes;
14     mask = mask >> 2;
15     non_zero_bytes = mask & 0x3;
16     ...
17   }
18 }
```

Listing 1.2. Pseudo code for AN encoded NS decompression. buffer is the input array while elements is the output array containing the uncompressed, decoded items. |buffer| denotes the array's number of elements.

Decompression and Decoding. Decompression and decoding is also straightforward. Listing 1.2 shows the according pseudo code. In this case, function `decompress` again receives an input and an output buffer and a loop iterates over the input buffer of AN-encoded and compressed data (lines 1,3). First, the compression mask is loaded, checked for errors and decoded (lines 5–7). Then, the buffer position is incremented (line 8), the number of non-zero bytes – denoted by the mask's lowest 2 bits – of the first data item is extracted (line 9) and the according bytes are stored (line 10). The restored item is checked against A and errors may be handled (line 11). Then, the decoded data item is stored in the output array and the read position of the input buffer is advanced by the number of non-zero bytes (lines 12,13). Finally, the mask is shifted right, so that the same steps can be repeated for the next three items, since always 4 items are represented by a single-byte compression mask (lines 14–16).

4.2 AN-encoded Run-Length Compression

The basic idea of Run-Length Compression (RLE) is to compress consecutive sequences of a same value – the *runs*. For compression, the distinct original value is stored together with the number of uninterrupted appearances – the *run length*. For decompression, these values are rolled out again.

General Idea

In contrast to our AN-encoded Null Suppression compression scheme, our AN-encoded RLE approach compresses first and encodes afterwards, since runs are

Final edited form was published in "Data Management Technologies and Applications 4th International Conference. Colmar 2015", S. 135-153. ISBN: 978-3-319-30162-4
https://doi.org/10.1007/978-3-319-30162-4_9

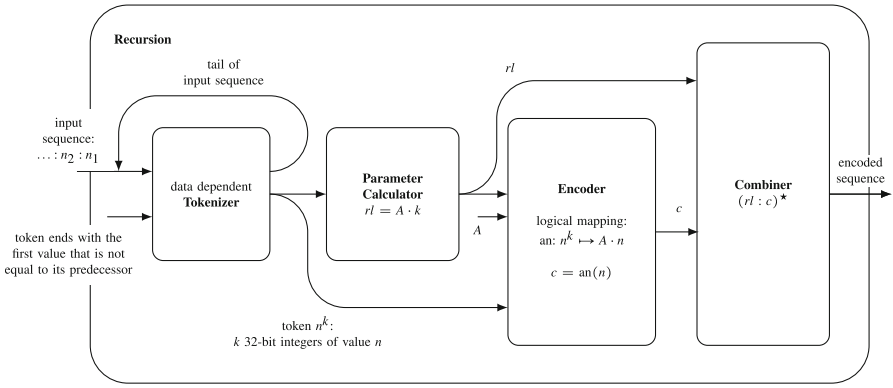


Fig. 2. Compression Scheme for Modularization for AN Run Length Encoding.

condensed to the value and its run length, therefore encoding only 2 values instead of long runs of values (see Fig. 2). That means, we reduce the necessary work for encoding using compression. In detail, AN-encoded RLE compression works as follows: Consecutive appearances of values are counted—the run lengths—using a *data dependent* `Tokenizer`. Whenever a new value is encountered, the previous value and its run length are encoded and written to the output. While the run-length is computed and encoded in `Parameter Calculator`, the value is encoded in the `Encoder`. The `Combiner` produces the resulting AN-encoded RLE output sequence. Decompression is done by reading in pairs of encoded values and their encoded run lengths. After checking both of them against A the decoded value is written "run length" times to the output buffer.

SIMD-Based Implementation

The SIMD variants differ only in comparing multiple input values against the current one, for compression, and in writing out multiple values at once. Since the encoding and decoding only takes place on the single values and their run lengths, changes to the algorithm are actually the same as to the sequential variant.

4.3 Summary

As shown in this section, the combination of AN-encoding and compression schemes differ, whereas the combination is straightforward. Nevertheless, the combination is useful from a database perspective and the AN-encoding integrates seamless in efficient vectorized compression techniques. However, our two examples are only a starting point and further research is necessary to protect compressed data in an efficient way. Additionally, the parameterization of the AN-encoding approach has a high impact as presented in the next section which is also a open topic.

Final edited form was published in "Data Management Technologies and Applications 4th International Conference. Colmar 2015", S. 135-153. ISBN: 978-3-319-30162-4
https://doi.org/10.1007/978-3-319-30162-4_9

Table 1. Information about A: parameter A; $invA$: the inverse for 32-bit integers (if applicable); $|A|$: the number of effective bits of A; $p_1 \dots p_6$: the respective probabilities of not detecting $1 \dots 6$ bit flips; *NS comp. rate*: NS compression rate for 16-effective-bits random integers; *overhead*: memory overhead of AN-encoded compression compared to the unencoded compressed ratio.

A	inv(A)	$\|A\|$	p_1	p_2	p_3	p_4	p_5	p_6	NS comp. rate	NS overhead
compr.		-	-	-	-	-	-	-	0.561	-
3	2,863,311,531	2	0.0	14.2	3.74	2.73	1.124	0.567	0.729	30 %
5	3,435,973,837	3	0.0	7.2	3.31	1.73	0.890	0.439	0.762	36 %
13	3,303,820,997	4	0.0	2.2	1.72	0.93	0.515	0.282	0.793	41 %
26	—	5	0.0	2.2	1.72	0.93	0.515	0.282	0.803	43 %
59	2,693,454,067	6	0.0	0.0	0.51	0.34	0.210	0.130	0.808	44 %
118	—	7	0.0	0.0	0.51	0.34	0.210	0.130	0.810	
250	—	8	0.0	0.0	0.25	0.19	0.133	0.088	0.811	45 %
507	2,837,897,523	9	0.0	0.0	0.08	0.07	0.046	0.040	0.936	67 %
641	6,700,417	10	0.0	0.0	0.08	0.06	0.040	0.030	0.962	71 %
965	485,131,021	10	0.0	0.0	0.00	0.04	0.032	0.025	0.996	78 %
7567	3,745,538,415	13	0.0	0.0	0.00	0.00	0.007	0.007	1.054	88 %
58659	2,839,442,059	16	0.0	0.0	0.00	0.00	0.000	0.001	1.061	89 %

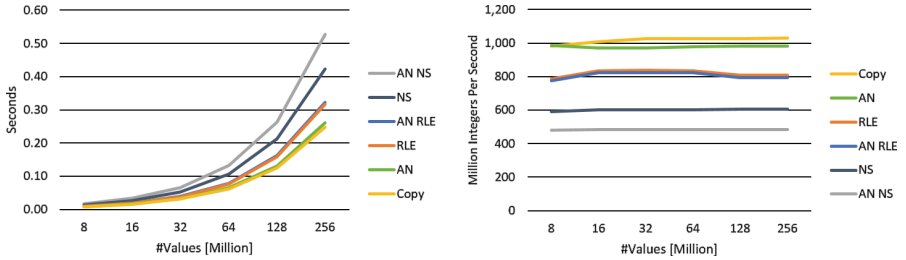
5 Evaluation

In this section, we first discuss the choice of the constant A and what trade-offs it introduces. Then, we show the experimental results of applying AN encoding to Null Suppression and Run Length Encoding using 32-bit integers with only 16 effective bits to guarantee compressibility. We present throughput measurements for both sequential and SIMD implementations using $A = 641$. The experiments were run on a machine with an ASUS P9X79 Pro mainboard running a 12-core Intel i7-3960X CPU and 8 X 4 GiB (32 GiB) DRAM on an Ubuntu 15.04 OS. Generally, our measurements were executed on different sizes of random data sets, in particular 8, 16, 32, 64, 128, and 256 Million integers. Since all encoding / compression is done by copying instead of in-place operations, we use copying as a baseline. To ensure that the compiler does not generate undesired SIMD code, we use the GCC compiler flag `-fno-tree-vectorize`.

5.1 An Encoding

As mentioned earlier, the choice of parameter A affects the fault detection rate, memory blow-up, and encoding/decoding performance. Table 1 shows some

Final edited form was published in "Data Management Technologies and Applications 4th International Conference. Colmar 2015", S. 135-153. ISBN: 978-3-319-30162-4
https://doi.org/10.1007/978-3-319-30162-4_9



(a) Runtime **(b)** MIPS

Fig. 3. Performance Evaluation of Sequential Algorithms.

"good" A's that range in their fault coverage[3], bit size and memory overhead, and whether there exists a multiplicative inverse for 32-bit arithmetic and thus fast decoding. For example, $A = 3$ has a size of 2 bits which leads to a 6% memory increase for 32-bit integers and we are able to detect all single bit flips but only 86% of double bit flips. On the other extreme, $A = 58,659$ ensures detecting up to 5 bit flips, but is 16 bits wide, leading to 50 % memory increase of 32-bit integers. In the end, the choice of A depends on how many bit flips should be detectable and how much redundancy is tolerable.

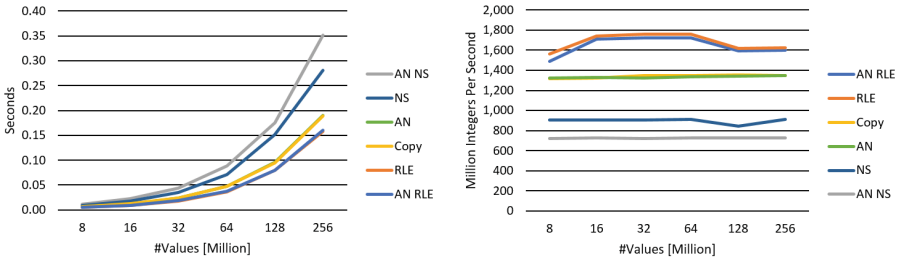
5.2 Compression Rates of AN-encoded Data Compression

AN-encoded Null Suppression. The last two columns of Table 1 show the typical compression rates for Null Suppression and the overhead of our AN encoding compared to pure NS. Notice that the original compression rate is about 0.561, and our AN-encoded NS scheme introduces a memory overhead of 30-89%. The overhead regarding uncompressed data reduces with smaller value ranges. For example, 16-bit compressible data using $A = 641$ occupies 3.8 % less space than uncompressed 32-bit data (0.962), while using ($A = 3$) occupies 27.1 % less space. The memory footprint of AN-encoded NS-compressed data exceeds uncompressed data when using an A which is more than 10 bits large.

AN-encoded Run Length Encoding. Since we assume 16 effective bits of data and $A = 641$ there is no actual memory overhead for RLE when comparing pure and AN-encoded RLE. If the values and run lengths are further compressed – e.g. using Null Suppression – then AN-coding incurs the overhead of the bits of the used A. RLE was tested with a fixed run length of 16.

[3] The probabilities for the table are taken from the experimental results of [11]; they can be found on https://www4.cs.fau.de/Research/CoRed/experiments.

Final edited form was published in "Data Management Technologies and Applications 4th International Conference. Colmar 2015", S. 135-153. ISBN: 978-3-319-30162-4
https://doi.org/10.1007/978-3-319-30162-4_9



(a) Runtime **(b)** MIPS

Fig. 4. Performance Evaluation of SIMD Algorithms.

5.3 Performance of Encoding / Compression

Figures 3 and 4 show the throughput measurements for our AN-encoded compression techniques for the sequential as well as SIMD (SSE 4.1) versions, respectively, using $A = 641$. In the experiments, we varied the size of the data sets. Table 2 lists the average number of MIPS, which is quit stable for all algorithms across the sizes of the sets.

Copying the data from one array to another is the baseline. AN-encoding itself leads to almost no overhead, which can be attributed to the good pipelining of simple multiplications. The SIMD RLE variant is faster than purely copying, because only an eighth of the original number of values is written: Instead of 16 32-bit integers, only 1 value and 1 run length, both 32 bits wide, are written.

For Null Suppression, AN-encoding incurs an overhead of $483/601 = 0.8020\,\%$ and $725/896 = 0.8119\,\%$ for sequential and SIMD variants, respectively. Next to the additional multiplication, the increased amount of data written to memory is responsible for the increase in runtime. The overhead for our AN-encoded RLE is $804/818 = 0.982\,\%$ and $1,639/1,677 = 0.982\,\%$ for sequential and SIMD variants, respectively. Depending on the run lengths, the multiplication for encoding is negligible, since much fewer ones than for NS are actually executed (one eighth, as described above).

5.4 Performance of Decoding / Decompression

Table 3 outlines the average MIPS for decoding / decompressing from the encoded / compressed formats. As expected, pure AN-coding exhibits (almost) the same results as encoding, since there exists a multiplicative inverse for $A = 641$ and the differences are negligible.

Decoding Null Suppression is slower than Compression, since more data is written back to memory than during decoding. AN-encoded NS decoding becomes much worse now, because every encoded value is first tested against A using the modulo operation and then decoded back by multiplying with the

Final edited form was published in "Data Management Technologies and Applications 4th International Conference. Colmar 2015", S. 135-153. ISBN: 978-3-319-30162-4
https://doi.org/10.1007/978-3-319-30162-4_9

Table 2. Average MIPS for encoding from raw data. Absolute and relative (compared to Copy) numbers are given.

	Copy		AN		NS		AN+NS		RLE		AN+RLE	
	Abs	Rel	Abs	Rel	Abs	Rel	Abs	Rel	Abs	Rel	Abs	Rel
Sequential	1,016	1.00	978	0.96	601	0.59	483	0.48	818	0.81	804	0.79
SIMD	1,339	1.00	1,333	1.00	896	0.67	725	0.54	1,677	1.25	1,639	1.22

Table 3. Average MIPS for decoding. Absolute and relative (compared to Copy) numbers are given.

	Copy		AN		NS		AN+NS		RLE		AN+RLE	
	Abs	Rel	Abs	Rel	Abs	Rel	Abs	Rel	Abs	Rel	Abs	Rel
Sequential	1,016	1.00	1,013	1.00	500	0.49	260	0.26	1,051	1.03	994	0.98
SIMD	1,339	1.00	1,317	0.98	750	0.56	570	0.43	1,258	0.94	1,339	1.00

inverse. Decoding overead is as high as $260/500 = 0.5248\,\%$ and $570/750 = 0.7624\,\%$ for sequential and SIMD code, respectively.

RLE decoding is very fast since on the one hand much less memory is read compared to what is read – due to the run length of 16 – and on the other hand unrolling of values is much simpler than NS decoding.

5.5 Summary

We find these results encouraging. The choice of parameter A provides trade-offs in terms of error detection capability, memory overhead, and performance penalty, while the speeds of the compression schemes are affected differently by the encoding overhead – both in terms of added complexity of code as well as dependency on the data characteristics.

6 Conclusion

Modern database systems are very often in the position to store their entire data in main memory. The reasons are manifold: (i) increased main memory capacities, (ii) column-oriented storage format and (iii) lightweight data compression techniques. Unfortunately, hardware becomes more and more vulnerable to random faults, so that e.g., the probability rate for bit flips in main memory increases, and this rate is likely to escalate in future dynamic random-access memory (DRAM) modules. Since the data is highly compressed by the lightweight compression algorithms, multi bit flips will have an extreme impact on the reliability of database systems. To overcome this issue, we have introduce our research on error resilient lightweight data compression algorithms in this paper. In detail, we have utilized arithmetic AN encoding, which is one family of codes which is an interesting candidate for effective software-based error

Final edited form was published in "Data Management Technologies and Applications 4th International Conference. Colmar 2015", S. 135-153. ISBN: 978-3-319-30162-4
https://doi.org/10.1007/978-3-319-30162-4_9

detection. We have presented two algorithms: (i) AN-encoded Null Suppression and (ii) AN-encoded Run Length Encoding. We have shown in our experiments that by using AN encoding, much higher bit flip detection capabilities are achievable than with SECDED ECC. Furthermore, Our evaluation indicates that data compression schemes augmented with AN encoding become resilient at a low memory and performance cost. As an example, a "golden" A of 641 makes 16-bit data completely resilient to single and double bit flips. Depending on the scenario and the compression schemes, AN encoding results in little to no performance penalties. Of course, encoding leads to memory overhead, but we also showed that gains over uncompressed data are still possible. For instance, AN encoded Null Suppression occupies 4% less space than uncompressed data, with a 5-10% slowdown of compression/decompression speed for the case of SIMD.

Acknowledgements. This work is partly supported by the German Research Foundation (DFG) within the Cluster of Excellence "Center for Advanced Electronics Dresden" (cfAED) and by the DFG-grant LE-1416/26-1.

References

1. Abadi, D., Madden, S., Ferreira, M.: Integrating compression and execution in column-oriented database systems. In: Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data, pp. 671–682 (2006)
2. Antoshenkov, G., Lomet, D.B., Murray, J.: Order preserving compression. In: Proceedings of the Twelfth International Conference on Data Engineering, ICDE 1996, pp. 655–663 (1996)
3. Bassiouni, M.A.: Data compression in scientific and statistical databases. IEEE Trans. Softw. Eng. **11**(10), 1047–1058 (1985)
4. Bohannon, P., Rastogi, R., Seshadri, S., Silberschatz, A., Sudarshan, S.: Detection and recovery techniques for database corruption. IEEE Trans. Knowl. Data Eng. **15**(5), 1120–1136 (2003)
5. Boncz, P.A., Manegold, S., Kersten, M.L.: Database architecture optimized for the new bottleneck: memory access. In: Proceedings of the 25th International Conference on Very Large Data Bases, VLDB 1999, pp. 54–65 (1999)
6. Chen, Z., Gehrke, J., Korn, F.: Query optimization in compressed database systems. SIGMOD Rec. **30**(2), 271–282 (2001)
7. Goldstein, J., Ramakrishnan, R., Shaft, U.: Compressing relations and indexes. In: Proceedings of 14th International Conference on Data Engineering, pp. 370–379, February 1998
8. Graefe, G., Kuno, H., Seeger, B.: Self-diagnosing and self-healing indexes. In: DBTest, pp. 8:1–8:8 (2012)
9. Graefe, G., Stonecipher, R.: Efficient verification of b-tree integrity. In: BTW, pp. 27–46 (2009)
10. Hildebrandt, J., Habich, D., Damme, P., Lehner, W.: Modularization of lightweight data compression algorithms. Technical report, Department of Computer Science, Technische Universität Dresden, November 2015. https://wwwdb.inf.tu-dresden.de/misc/team/habich/dcc2016.pdf. submitted to DCC 2016

Final edited form was published in "Data Management Technologies and Applications 4th International Conference. Colmar 2015", S. 135-153. ISBN: 978-3-319-30162-4
https://doi.org/10.1007/978-3-319-30162-4_9

11. Hoffmann, M., Ulbrich, P., Dietrich, C., Schirmeier, H., Lohmann, D., Schröder-Preikschat, W.: A practitioner's guide to software-based soft-error mitigation using AN-codes. In: HASE 2014, pp. 33–40 (2014)
12. Huffman, D.A.: A method for the construction of minimum-redundancy codes. Proc. Inst. Radio Eng. **40**(9), 1098–1101 (1952)
13. Hwang, A.A., Stefanovici, I.A., Schroeder, B.: Cosmic rays don't strike twice: understanding the nature of DRAM errors and the implications for system design. SIGARCH Comput. Archit. News **40**(1), 111–122 (2012)
14. Kissinger, T., Kiefer, T., Schlegel, B., Habich, D., Molka, D., Lehner, W.: ERIS: a numa-aware in-memory storage engine for analytical workload. In: International Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures - ADMS, pp. 74–85 (2014)
15. Kolditz, T., Kissinger, T., Schlegel, B., Habich, D., Lehner, W.: Online bit flip detection for in-memory b-trees on unreliable hardware. In: DaMoN, pp. 5:1–5:9 (2014)
16. Lehman, T.J., Carey, M.J.: Query processing in main memory database management systems. In: Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data, SIGMOD 1986, pp. 239–250 (1986)
17. Lehner, W.: Energy-efficient in-memory database computing. In: Design, Automation and Test in Europe, DATE 13, Grenoble, France, 18–22 March 2013, pp. 470–474 (2013)
18. Lemire, D., Boytsov, L.: Decoding billions of integers per second through vectorization. CoRR abs/1209.2137 (2012)
19. May, T.C., Woods, M.H.: Alpha-particle-induced soft errors in dynamic memories. IEEE Trans. Electron Devices **26**(1), 2–9 (1979)
20. Moon, T.K.: Error Correction Coding: Mathematical Methods and Algorithms. Wiley, Hoboken (2005)
21. Reghbati, H.K.: An overview of data compression techniques. IEEE Comput. **14**(4), 71–75 (1981)
22. Roth, M.A., Van Horn, S.J.: Database compression. SIGMOD Rec. **22**(3), 31–39 (1993)
23. Schiffel, U.: Hardware Error Detection Using AN-Codes. Ph.D. thesis, Technische Universität Dresden (2011)
24. Schlegel, B., Gemulla, R., Lehner, W.: Fast integer compression using simd instructions. In: DaMoN. pp. 34–40 (2010)
25. Schroeder, B., Gibson, G.A.: A large-scale study of failures in high performance-computing systems. Dependable Secure Comput. **7**(4), 337–350 (2010)
26. Schroeder, B., Pinheiro, E., Weber, W.D.: Dram errors in the wild: a large-scale field study. In: Proceedings of the Eleventh International Joint Conference on Measurement and Modeling of Computer Systems, SIGMETRICS 2009, pp. 193–204 (2009)
27. Stepanov, A.A., Gangolli, A.R., Rose, D.E., Ernst, R.J., Oberoi, P.S.: Simd-based decoding of posting lists. In: Proceedings of the 20th ACM International Conference on Information and Knowledge Management, CIKM 2011, pp. 317–326 (2011)
28. Stonebraker, M.: Technical perspective - one size fits all: an idea whose time has come and gone. Commun. ACM **51**(12), 76 (2008)
29. Sullivan, M., Stonebraker, M.: Using write protected data structures to improve software fault tolerance in highly available database management systems. In: VLDB, pp. 171–180 (1991)
30. Warren, H.S.: Hacker's Delight. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)

Final edited form was published in "Data Management Technologies and Applications 4th International Conference. Colmar 2015", S. 135-153. ISBN: 978-3-319-30162-4
https://doi.org/10.1007/978-3-319-30162-4_9

31. Witten, I.H., Neal, R.M., Cleary, J.G.: Arithmetic coding for data compression. Commun. ACM **30**(6), 520–540 (1987)
32. Ziv, J., Lempel, A.: A universal algorithm for sequential data compression. IEEE Trans. Inf. Theor. **23**(3), 337–343 (1977)
33. Zukowski, M., Heman, S., Nes, N., Boncz, P.: Super-scalar ram-cpu cache compression. In: Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, pp. 59–59, April 2006