Resiliency-aware Data Compression for In-Memory Database Systems

Till Kolditz*, Dirk Habich*, Patrick Damme*, Wolfgang Lehner*, Dmitrii Kuvaiskii†, Oleksii Oleksenko† and Christof Fetzer†

*Technische Universität Dresden, *Database Systems Group, †Systems Engineering Group, Dresden, Germany*

{*firstname.lastname*}*@tu-dresden.de*

Keywords: in-memory database systems, data integrity, lightweight data compression, an encoding

Abstract: Nowadays, database systems pursuit a main memory-centric architecture, where the entire business-related data is stored and processed in a compressed form in main memory. In this case, the performance gain is massive because database operations can benefit from its higher bandwidth and lower latency. However, current main memory-centric database systems utilize general-purpose error detection and correction solutions to address the emerging problem of increasing dynamic error rate of main memory. The costs of these general-purpose methods dramatically increases with increasing error rates. To reduce these costs, we have to exploit context knowledge of database systems for resiliency. Therefore, we introduce our vision of resiliency-aware data compression in this paper, where we want to exploit the benefits of both fields in an integrated approach with low performance and memory overhead. In detail, we present and evaluate a first approach using AN encoding and two different compression schemes to show the potentials and challenges of our vision.

1 Motivation

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 (Hwang et al., 2012; Italiano, 2010). 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 the dynamic error rate is still quite low, it is predicted to increase substantially in the near future (Hwang et al., 2012; Italiano, 2010). Moreover, dynamic errors already have a significant impact on large-scale application on massive data sets. The field of fault tolerance against dynamic memory errors is not new and several techniques are well-known. A general applicable approach is executing the same computation multiple times. In this case, any dynamic error can be detected by comparing the final results. 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 to the data (Moon, 2005). Regarding DRAM bit flips, the most commonly used approach is hardware-based (72,64)-Hamming ECC (Moon, 2005). It realizes single-error correction and double-error detection. Many other general coding algorithms are available, whereas the enhanced coding schemes are more robust, however their coding results in higher redundancy 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.

Based on the significant developments in the hardware sector, database systems have initiated a major shift from disk-centric to main memory-centric architectures. Servers with terabytes of main memory are available for a reasonable price, where the entire data pool can be kept completely in main memory. As shown in different papers (Abadi et al., ; Chen et al., 2001; Lemire and Boytsov, 2012), the performance gain is massive because database operations benefit from its higher bandwidth and lower latency (Garcia-Molina and Salem, 1992). However, current database systems do not efficiently address the emerging problem of random dynamic faults.

In order to tackle this issue, we propose to tightly combine existing techniques form the field of error detecting codes and data compression in an appropriate way: *resiliency-aware data compression techniques*. Data compression techniques play an important role in main memory-centric database systems. These techniques are employed to enable keeping and processing all business-related data in main memory. As can be imagined, compressed data is very sensitive to dynamic errors and an increased dynamic error rate as described above has a high impact on systems relying on compressed data. Therefore, we focus on the resilience aspect of compressed data. In our vision, the benefits of compression should not be completely neutralized by error detection mechanisms. To reduce the overall overhead, query processing on resiliency-aware compressed data should be possible directly without explicitly decompressing and re-encoding the data. Finally, error detection should be possible in an online fashion, so that wrong results can be excluded to a well-defined degree. To summarize, we want to replace data compression techniques with *resiliency-aware data compression* techniques to retain as many positive effects of data compression as possible, but extend it with specific resiliency aspects to efficiently handle an increasing dynamic error rate.

To show the potentials and challenges of our vision, 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 workloads. From the data compression domain, we decided to use two prominent lightweight techniques: Null Suppression (Abadi et al., ; Roth and Van Horn, 1993), and Run Length Compression (Abadi et al.,). As we are going to show, the combination approach differs and depends on various factors. Furthermore, we provide an analysis on how the basic parameter A of AN encoding can be chosen to detect various amounts of bit flips with very low performance penalties and low memory overhead.

The remainder of the paper is structured as follows: In Section 2, we give a detailed insight into AN encoding. Based on this description, we introduce our AN-encoded extension for Null Suppression and Run-Length Compression in Section 3. Next, we present our evaluation in Section 4, where we discuss the parameterization of AN encoding and provide throughput comparisons of our AN-encoded compression schemes. Finally, we conclude the paper with related work in Section 5 as well as a conclusion and future work in Section 6.

2 AN-Encoding Technique

To tackle our vision of *resiliency-aware data compression techniques*, we decided to utilize AN encoding as our resilience technique in a first step. AN encoding is a technique to detect data corruption caused by transient (e.g., dynamic bit flips) and permanent (e.g., stuck-at-1) hardware faults (Schiffel, 2011). Furthermore, AN encoding offers some features that we assume beneficial for database system, in particular for efficient query processing as described later.

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:

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

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:

$$\hat{n} \bmod A = 0$$

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

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

Beneficial Features of AN encoding

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.

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 tradeoffs 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 (Hoffmann et al., 2014). 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 is negligible during encoding (since multiplication requires only $2 - 3$ CPU cycles), 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 (Warren, 2002).

3 Resiliency-aware Data Compression

Lightweight data compression techniques like dictionary or run-length compression play an important role in main memory database systems. These techniques are employed to enable holding and processing all business-related data in main memory, whereas all compression techniques in database systems are lossless. The general approach of data compression is to encode data using fewer bits than the original representation. In contrast to heavyweight compression techniques like Huffmann, lightweight techniques are less compute-intensive, whereas they utilize context knowledge to achieve a good compression rate.

To best of our knowledge, nowadays no additional information is added to detect bit flip corruption of compressed data in main memory database systems. In order to tackle an increasing bit flip error rate, in particular for dynamic errors, we want to tightly combine techniques from both fields of lightweight data compression and resilience techniques like AN encoding. One the one hand, lightweight data compression eliminate data redundancy to represent data using fewer bits. On the other hand, resilience techniques introduces data redundancy to detect bit flips. Therefore, both fields have opposed aims and the combination approach have to carefully designed, so that the benefits of both fields remains. 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 Section 4.1 and (ii) AN-encoded Run-Length Compression in Section 4.2.

3.1 AN-encoded Null Suppression

Null Suppression (NS) is the most well-studied kind of lightweight compression. 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 of. 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.

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. (Schlegel et al., 2010) 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 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. For bitwise or byte-wise NS, the only reasonable way to tightly combine NS and AN encoding is to encode the original value first and to compress afterwards. Since the information about eliminated bits or bytes is stored in the compression mask, we would compute the leading zero bits/bytes twice in any other combination approach. This incurs unnecessary additional computational overhead, which we avoid in our approach.

3.1.1 Encoding and Compression

Encoded compression for NS works as follows. Listing 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 (lines 11 and 12). Finally, the mask and the compressed encoded words are stored in the output buffer (lines 13-16). 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.

3.1.2 Decompression and Decoding

Decompression and decoding is also straightforward. In this case, a loop iterates over the input buffer of AN-encoded and compressed data. First, the compression mask is loaded. Then, the number of non-zero bytes – denoted by the mask's lowest 2 bits – of the first data item is stored and the according bytes are stored. The restored item is checked against A and errors may be handled. 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. 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.

```
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)       ↩
12         | (z2 << 2) | z1;
13     buffer ← mask;
14     buffer ← n1;
15     ...
16     buffer ← n4;
17   }
18 }
```

Listing 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.

3.2 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.

In contrast to our AN-encoded Null Suppression compression scheme, our AN-encoded RLE approach compresses first and encodes afterwards, since runs are condensed to the value and its run length, therefore encoding only 2 values instead of long runs of values. 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. Whenever a new value is encountered, the previous value and its run length are encoded and written to the output buffer. 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. The SIMD variant works analogously by comparing against and writing out multiple values at once

A	$\|A\|$	p_1	p_2	p_3	p_4	p_5	p_6	cyc	Null Sup. comp. rate	AN-encoded mem. overhead
compr.	-	-	-	-	-	-	-	-	0.561	-
3	4	0.0	14.2	3.74	2.73	1.124	0.567	8	0.729	29.94%
26	5	0.0	2.2	1.72	0.93	0.515	0.282	8	0.803	43.14%
118	7	0.0	0.0	0.51	0.34	0.210	0.130	8	0.810	44.39%
250	8	0.0	0.0	0.25	0.19	0.133	0.088	8	0.811	44.56%
507	9	0.0	0.0	0.08	0.07	0.046	0.040	8	0.936	66.84%
641	10	0.0	0.0	0.08	0.06	0.040	0.030	3	0.962	71.48%
7567	13	0.0	0.0	0.00	0.00	0.007	0.007	8	1.054	87.88%
58659	16	0.0	0.0	0.00	0.00	0.000	0.001	8	1.061	89,13%

Table 1: Choice of constant A with its number of effective bits $|A|$. $p_1 \dots p_6$ are the respective probabilities of not detecting $1 \dots 6$ bit flips, *cyc* – CPU cycles for 4-wise SIMD decoding on Ivy Bridge architecture (taken from Intel Intrinsics Guide: `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`), *comp. rate* – NS compression rate for 16-bit random integers – memory overhead of AN-encoded compression

for compression and decompression, respectively.

4 EVALUATION

In this section, we discuss first the choice of the constant A and what trade-offs it introduces. Then, we show experimental results of our AN-encoded extensions of Null Suppression and Run-Length Compression. Our evaluation is based on a set of 100 million 32-bit random integers with only 16 effective bits to guarantee compressibility and to have no overflow when encoding. The experiments were run on a machine with an ASUS P9X79 Pro mainboard running a 12-core Intel i7-3960X CPU and 8x4 GiB (32GiB) DRAM. The compression speed results are averages of 10 runs; the relative standard error does not exceed 0.05%. Due to space constraints, we only report our evaluation results of our AN-encoded Null suppression technique.

Parameterization of AN Encoding

As mentioned earlier, the choice of the constant A affects fault detection rate, memory blow-up, and encoding/decoding performance. Table 1 shows some "good" A's that range in their fault coverage, bit size, and decoding performance[1]. For example, $A = 3$ has a size of 2 bits, can detect all single bit flips but only 86% of double bit flips, and requires 4 CPU cycles for decoding. On the other extreme, $A = 58,659$ can detect up to 5 bit flips, but is 16 bits wide, leading to $2x$ memory increase. The "golden" A is equal to 641, with sufficiently high fault coverage, limited memory increase, and very fast decoding of only 3 cycles.

[1]The probabilities for the table are taken from the experimental results of (Hoffmann et al., 2014); they can be found on `https://www4.cs.fau.de/Research/CoRed/experiments`.

AN-encoded Null Suppression

The second last column of Table 1 shows the compression rates for our AN-encoded Null Suppression, whereas the last column reports the memory overhead. Notice that the original compression rate without AN-encoding is 0.561 as depicted in the first row. As we can see, our AN-encoded extension introduces a memory overhead of 30-90%. This overhead reduces with smaller value ranges. For our golden $A = 641$, our AN-encoded compressed data requires less space than uncompressed data. That means, we are able to compress and to protect our data with less space as uncompressed, which is a very good results from our point of view.

To evaluate the performance, we implemented our presented AN-encoded schemes in a sequential and a vectorized (SIMD - SSE 4.1) version, whereas we used $A = 641$ in our evaluation. As expected, AN-coding causes slowdowns of 20-25% in case of the sequential implementation, and 5-10% for the SIMD implementation. The difference between the sequential and SIMD slowdowns stems from the significantly different sequences of generated instructions; in a nutshell, the sequential version introduces four additional instructions while the SIMD version adds only one. Nevertheless, the performance slowdown is low, so that the combination of AN-encoding and compression is very promising to efficiently tackle an increasing dynamic bit flip rate.

5 Related Work

Several techniques have been presented to deal with certain error classes in the past. To our best knowledge, no research was done in the field of databases to protect in-memory data against arbitrary

dynamic bit flips, except our own investigations on error detecting B-Trees (Kolditz et al., 2014). In the field of databases, other 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, (Graefe and Stonecipher, 2009; Graefe et al., 2012) harden the well-known B-Tree and variants against errors during I/O-operations or against certain other tree corruptions, whereas the techniques are not suited for online error detection. Dynamic bit flips may lead to false positives and false negatives when querying such trees between there maintenance checks.

(Sullivan and Stonebraker, 1991) 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.

6 Conclusion

Low-cost, high-density main memory lead to a change in the database ecosystem towards in-memory designs, whereas data compression plays an important role, too. However, due to the constant increase of memory density, main memory becomes more and more unreliable. This results in dynamic random errors and in future hardware generations the error rates are likely to escalate. General purpose ECC mechanisms are not the appropriate solution from a cost perspective. Therefore, we propose to exploit database context knowledge by a tightly combination of compression and resilience techniques. As presented, AN-encoding is a code family which allows arithmetic operations on code words and, by that, is suitable for transactional and analytical workloads. We showed that by using AN-encoded extensions of compression schemes, much higher bit flip detection capabilities are achievable than with SECDED ECC. Our preliminary evaluation shows that data compression schemes augmented with AN-encoding become resilient at a low memory and performance cost. Nevertheless, more research in this direction is necessary from a conceptual as well as implementation perspective.

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

Abadi, D., Madden, S., and Ferreira, M. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, pages 671–682.

Chen, Z., Gehrke, J., and Korn, F. (2001). Query optimization in compressed database systems. *SIGMOD Rec.*, 30(2):271–282.

Garcia-Molina, H. and Salem, K. (1992). Main Memory Database Systems: An Overview. *Knowledge and Data Engineering*, 4(6).

Graefe, G., Kuno, H., and Seeger, B. (2012). Self-diagnosing and self-healing indexes. In *DBTest*, pages 8:1–8:8.

Graefe, G. and Stonecipher, R. (2009). Efficient verification of b-tree integrity. In *BTW*, pages 27–46.

Hoffmann, M., Ulbrich, P., Dietrich, C., Schirmeier, H., Lohmann, D., and Schröder-Preikschat, W. (2014). A Practitioner's Guide to Software-based Soft-Error Mitigation Using AN-Codes. In *HASE '14*, pages 33–40.

Hwang, A. A., Stefanovici, I. A., and Schroeder, B. (2012). Cosmic Rays Don't Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design. *SIGARCH Comput. Archit. News*, 40(1).

Italiano, G. F. (2010). Resilient algorithms and data structures. In *CIAC 2010*, pages 13–24.

Kolditz, T., Kissinger, T., Schlegel, B., Habich, D., and Lehner, W. (2014). Online bit flip detection for in-memory b-trees on unreliable hardware. In *DaMoN*, pages 5:1–5:9.

Lemire, D. and Boytsov, L. (2012). Decoding billions of integers per second through vectorization. *CoRR*, abs/1209.2137.

Moon, T. K. (2005). *Error Correction Coding: Mathematical Methods and Algorithms*. Wiley.

Roth, M. A. and Van Horn, S. J. (1993). Database compression. *SIGMOD Rec.*, 22(3):31–39.

Schiffel, U. (2011). *Hardware Error Detection Using AN-Codes*. PhD thesis, Technische Universität Dresden.

Schlegel, B., Gemulla, R., and Lehner, W. (2010). Fast integer compression using simd instructions. In *DaMoN*, pages 34–40.

Sullivan, M. and Stonebraker, M. (1991). Using write protected data structures to improve software fault tolerance in highly available database management systems. In *VLDB*, pages 171–180.

Warren, H. S. (2002). *Hacker's Delight*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.