Lossless Generic Numeric Time Series Encoding

Ilari Shafer
ishafer@cs.cmu.edu

18 April 2013

Abstract

Data analysis systems that are specialized for certain types of data can take advantage of properties of the data and queries to become more efficient. One type, *numeric time series*, has a distinct structure and range-based query patterns, but many current data representations and compression techniques in time series datastores do not optimize for these properties.

Based on a diverse time series archive and some properties of time series, we provide initial evidence that a fixed-point representation is suitable for many numeric time series. Using this representation, our preliminary experiments show that a simple technique based on Huffman coding the bit length of values can be amenable to range-based random access while providing compression ratios that are better than a general-purpose encoder on many datasets, but significantly worse on some. Delivering consistently superior performance will likely require optimizing bit-level operations.

1 Introduction

Efficient large-scale data archival and analysis is a source of recent interest. We have long since entered an era where maintaining a wide variety of operational data is affordable, and recent analysis platforms are stepping up to the challenge of delivering insight from massive archives of system logs [2], web page connectivity graphs [10], web content [14], and so forth.

A key feature of many of these analysis platforms is specialized data encoding. Reducing the size of archived data is clearly valuable for reducing storage footprint, but also (and often, more importantly) can accelerate queries. By reducing the number of bytes that must be scanned off slow disks, kept in cache, and sent over the network, lightweight compressors like LZO and Snappy [6] optimize for the bandwidth limitations on- and off-chip; their use is now standard practice in many systems.

Effective compression should not simply be a matter of picking one well-known compressor. Suitable compression techniques are very data-dependent, particularly from the perspective of query acceleration [1]. Simple encodings such as dictionary coding and run-length encoding can be very inexpensive CPU-wise, but are only effective if there are few distinct values or runs, respectively. As an example of a more complex data-dependent encoding, the the zip-block representation in GBase [10] is optimized for graph data.

In this project, we[1] consider one particular type of data—numeric time series—that has a distinct structure as compared to other types of data. A few examples of numeric time series are stock prices, systems monitoring streams, ECG recordings, and audio signals, all of which have been stored and analyzed in various time series archival systems. Some of these time series have specialized compressors, including model-based ECG representations and audio audio codecs such as MP3 and FLAC.

Nonetheless, many do not, and yet there is interest in storing a variety of these time series *losslessly* in a reasonably general-purpose time series database. To that end, we briefly survey current methods used in

[0]These first two sections are mostly the same as the project proposal
[1]The "editorial we"

research and practice in Section 2, describe some processing demands and general properties of numeric time series data in Section 3, and describe our compression approach in Sections 4-5. We evaluate these methods in Section 6, and describe a few directions in light of the results in Section 8.

2 Compression in today's time series datastores

By and large, today's time series datastores use "generic," widely-available compressors, many of which are designed or optimized for textual data. Table 1 surveys a variety of archival systems and the compression techniques they use; most are based on Lempel-Ziv with little attention to the entropy coder.

System	Supported compression methods
OpenTSDB [19]	LZO, GZip, Snappy + limited delta
DataGarage [13]	DGZip (careful bit-packing + LZ), Zip
Saturnalia DB [15]	limited variable width + ZigZag
Dremel [14]	Unspecified general-purpose encoder
TSDS [20]	DEFLATE, Szip (from underling HDF5 storage)
tsdb [5]	QuickLZ
Vertica [11]	RLE, delta, LZO, "integer packing" [7]
FinanceDB	GZip, unknown proprietary encoder
RRDtool [16]	None

Table 1: Compression techniques used by a selection of time series databases.

Some systems use such compressors (many designed for text) since they are simply layers on top of other compressed formats: for example, TSDS [20] is built on HDF5 files and OpenTSDB [19] is constrained to the data layout and encoding options in its underlying HBase cluster. Even a highly optimized financial database ("FinanceDB") uses gzip to compress data in its historical store.

Nonetheless, some glimmers of evidence have started to emerge that indicate a much more specialized approach would be more effective in optimizing query speed and storage space. For example, FinanceDB also offers a (unknown and proprietary) less aggressive but much faster compression option, and research efforts to store systems monitoring data have started to demonstrate the benefits of customized compression [13].

3 Data demands and properties

Our goals in designing or selecting an encoder specialized for numeric time series data are twofold:

Lossless: It should encode data losslessly. Although many individual applications can tolerate loss up to the bound of the consumer (e.g., lossy audio codecs that remove data past the range of human perception), the type of information that can be dropped is not clear a priori across a broad archive.

Randomly accessible: To eventually support fast range queries, compressed streams should accomodate random access—it is desirable to be able to start decoding from any point in the stream.

To guide the choice of encoder, we exploit two general properties that hold across a variety of time series measurements. Again, having more knowledge about the data (such as a predictive model) can lead to more effective domain-specific techniques, but limits the generality of the resulting technique.

Smoothness: Time series are often smooth over time: many data sources are sampled from an underlying source that is not random. Even data that varies in a notoriously unpredictable fashion, such as stock price data, often does so in a fairly limited range.

Fixed precision: Many numeric time series data sources naturally have a fixed precision. By way of a few examples, raw sensor data is necessarily quantized by analog-to-digital converters, and computer systems metrics are frequently representations of some integer counter.

To substantiate these properties and evaluate the effectiveness of the methods described in this report, we use data from the UCR Archive [18], a diverse repository of numeric time series data. The distribution we consider consists of 30 "major" datasets from a variety of sources (e.g., motion capture, stocks, medical, power, environmental, sensors). The data has been used for a variety of separate papers and is in a collection of different formats, most of which are some form of delimited text. In an effort to be representative, we parse as much of the archive as is feasible within the time constraints of this project, a process which excludes:

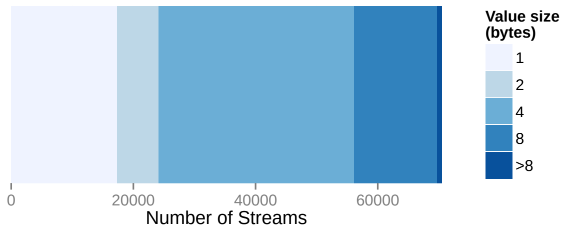
1. A synthetic dataset used for testing anomaly detection
2. Portions of the archive that are in Excel format (≈0.6% of the archive by size)
3. A dataset with no clear transformation to time series (≈0.004% of the archive by size)

We refer to the corpus without these three items as "the archive" in the remainder of this report. Our evaluation runs over the portion of the archive we can represent uniformly, which we discuss in the next section.

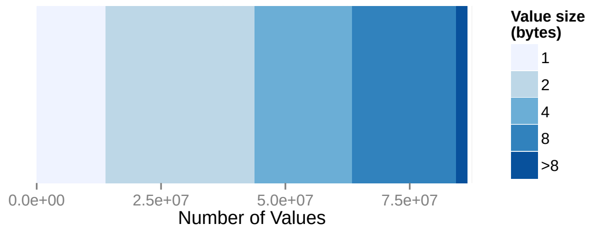
4 Value representation

A first question for an encoder is the choice of how values will be represented. The straightforward answer is to encode integers as integers and real numbers as floating-point values. Based upon the properties described above, we instead select a decimal fixed-point representation that uses a single exponent per stream and stores the mantissas.

It is not immediately obvious that we can represent all measurements in a streams with a single exponent. That we can is something of a product of the smoothness of timeseries data: the dynamic range of many of the observational datasets in the archive is fairly small. Figure 1 substantiates this claim with data from the archive. The bar charts are weighted by number of streams and number of observations, respectively, and it is evident that a significant fraction of the archive is quite limited in range (lighter blue portions). Admittedly, the archive is only one source of time series data, but it is worth noting that a few other datasets (sensor and systems monitoring) not included in this evaluation are also integral.



(a) Breakdown by number of value streams



(b) Breakdown by number of measurements

Figure 1: Percentage of the archive representable with a fixed exponent

One reason that fixed-point is unpopular for many applications, and a primary reason that computers and signal processors use floating-point representations instead of fixed-point, is that intermediate results rapidly become imprecise. However, our focus is on storing raw measurements; any consumer of the data would convert it to floating point before performing such operations.

This representation is essentially a variable-per-stream-width binary format; the size of the resulting corpus afterwards is about 2x smaller than the original archive. We use these binary streams as a source of data and baseline for comparison with a selection of encoding techniques, which are described in the next section.

5 Encoding techniques

We compare four encoding schemes: a simple scheme we designed that uses a Huffman coder to exploit limited exponent range, two universal integer codes, and a very popular general-purpose compression library. The first three were built for this project, and use the same underlying bitstream. Since the project source is a bit too lengthy to print (≈1500 SLOC of C++, 500 SLOC of Python), we have made it available at `https://github.com/mrcaps/generic-ts-encoding`.

For all techniques, rather than encode fixed-point values directly, we encode the differences ("deltas") between values. This is once again an optimization for the smoothness of the datasets in the archive: in the archive (as well as other time series data we have observed), deltas are smaller and often less unique than the values themselves. We support this choice in our evaluation in Section 6.

5.1 Log-Huffman

The initial proposal for this project suggested using Huffman encoding numeric values. While we could apply this technique bytewise, it clearly fails to scale to entire 8-byte deltas (unless we use an unreasonably large dictionary). Instead, we observe that therefore the range of their logs base 2 (i.e., the number of bits required to represent the value) is also small, and more importantly, often highly skewed. To get a sense of this Figure 2 shows the distribution of bits required for deltas for a selection of datasets in the archive.

The "Log-Huffman" encoder is designed to optimize for this skew. It Huffman codes the number of bits required to send a value, then sends the value itself less the first bit (since that bit must have been a 1). The stream looks like:

header	Code 1	Value 1	⋯

The "header" shown above is the generated Huffman tree packed onto the beginning of the bitstream. Since the alphabet (exponents) is not known by the decoder, and is not always contiguous, we cannot simply send the codeword lengths. Instead, we perform a preorder traversal on the tree, prepending the value of a node in the traversal with a 0 if it is an internal node or a 1 if it is a leaf. Since the alphabet is small in range, we send each exponent as an offset from a fixed base value that starts the stream. Denoting the alphabet as the set E, we send the tree (the header) as:

$\min E$	$\log_2(\max E - \min E)$	$0 \vert 1$	⋯

where $\min E$ is the minimum exponent in a fixed 7 bits, $\log_2(\max E - \min E)$ is the number of bits required for each offset (also sent in a fixed 7 bits), and each following entry is either 0 or 1 followed by an exponent sent in $\log_2(\max E - \min E)$ bits. Assuming there are at least two exponents, the number of bits required to send the dictionary is then $14 + |E|\left(\lceil \log_2(\max E - \min E)\rceil + 2\right) - 1$, somewhat smaller than the naïve $14 + 9|E| - 1$ bits.

For this technique and the next two, since the encoding can only represent positive integers, we ZigZag encode[2] the original signed

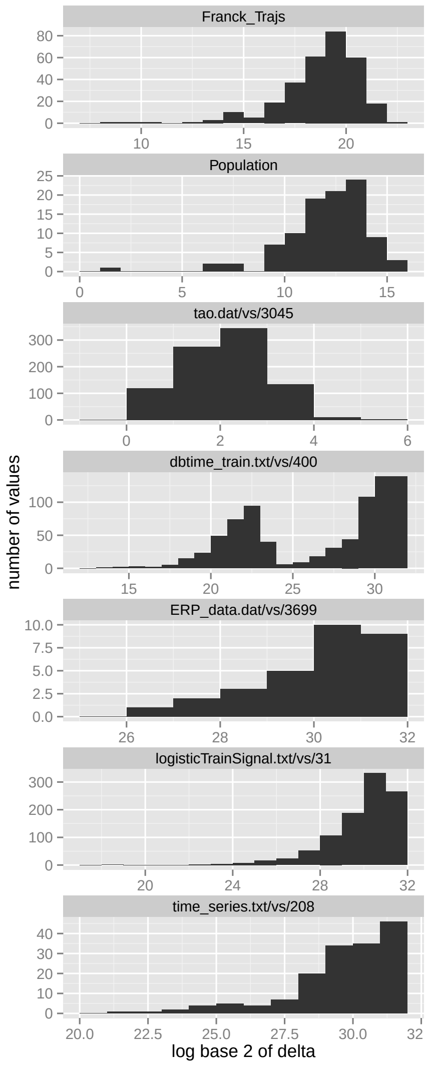


Figure 2: Distribution of delta mantissa size for a few sample streams

[2]`https://developers.google.com/protocol-buffers/docs/encoding`

integer input and add 1. For the reader interested in more of the implementation, it is available in `src/compressor/loghuffman.hpp`[3].

5.2 Elias gamma

If we do not assume or compute a distribution of bit lengths, we can use one of the multiple universal codes for integers. This is a useful point of comparison for both speed and space efficiency. These codes represent integers of any size with unique prefix codes.

One of the most well-known is Elias gamma coding, which simply prefixes a value with a unary encoding (with zeros) of its number of bits less one. For example, the code for 1 is 1 (1 bit to represent), the code for 4 is 00100, and the code for 15 is 0001111. While this code can represent small values effectively, the unary string becomes long for large values. We implement this code and use it for comparison.

5.3 Elias delta

For values that have a broader distribution (as the ones we consider do), the Elias delta code reduces the overhead of sending the unary string by using an Elias gamma code instead of a unary encoding for sending the bit length of the value. This is perhaps easiest to grasp through a few lines of (nearly verbatim) code from our implementation, where `nb` is the number of bits required to represent the value `v`, and `nb_nb` is the number of bits for `nb`:

```
bitstream.write_bits(0, nb_nb-1); //second argument is the number of bits to write
bitstream.write_bits(nb, nb_nb);
bitstream.write_bits(v & ~(1 << nb), nb-1); //~(1 << nb) masks off leading 1
```

As the reader might wonder, we can continue this process of Elias gamma coding bit lengths recursively—this is the Elias omega code. We do not compare Elias omega codes.

5.4 zlib

We would like a point of comparison with a "generic" compression technique used in current time series storage solutions. For this purpose, we use the zlib library, which is an LZ77 compressor with a Huffman entropy coder. This is the same technique used by gzip. We use the default compression settings and encode each stream in a single invocation of the compressor.

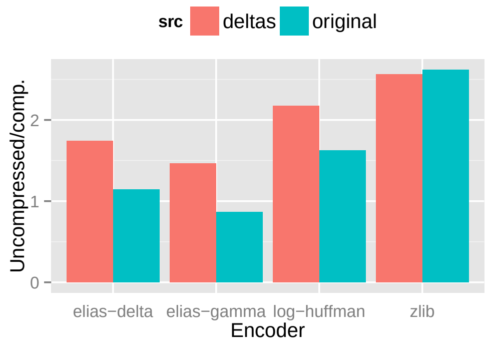
6 Results and discussion

Our experiments compare how effectively the above techniques can compress time series data in terms of both encoding speed and space. All experiments were performed on a physical machine with a single Intel Core i7 860 CPU @2.80GHz and 12GB of DDR3 RAM running Windows 7 Enterprise. The compression is memory-to-memory, from input that has been just touched sequentially by the delta encoder. Outside of the timed block we check the decoded output against the original input; despite fixing many bugs, there remains one that affects the Elias gamma encoder on 0.039% (by size) of the archive. For a baseline, we use the per-stream variable-width size of fixed-point data (i.e., 1, 2, 4, or 8 bytes).

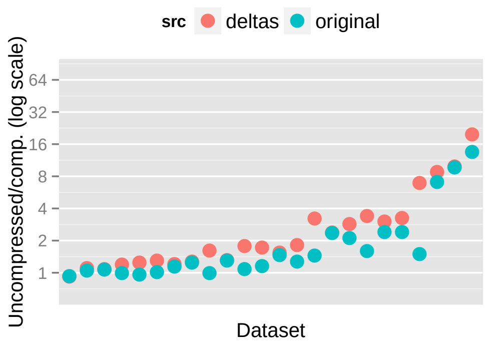
Encoding deltas: Our first experiments evaluate whether delta encoding is effective at increasing compressibility. We run all encoders over the archive both with and without delta encoding, and compare the resulting

[3] `https://github.com/mrcaps/generic-ts-encoding/blob/master/compressors/src/compressor/loghuffman.hpp`

compressed sizes. Figure 3a shows the mean compression ratio (over streams) for each encoder; there is visible benefit from storing deltas for the entropy coders. The benefits are fairly consistent across datasets, as shown in Figure 3b (where datasets are sorted by their mean compression ratio).



(a) By encoder (means over all datasets)



(b) By dataset (means over all encoders)

Figure 3: Increased compressibility of deltas as opposed to original data.

Space: We break down the compression ratio obtained by each encoder by dataset; the result is shown in Figure 4. Again, datasets are sorted by mean compression ratio. Elias gamma encoding is consistently the worst encoder from the perspective of space, and although Elias delta coding fares better it is typically inferior to zlib. Log-Huffman shows more promise: on datasets where zlib does not perform particularly well, it often achieves a better ratio. However, on a few of the highly redundant datasets at right, zlib typically performs much better.

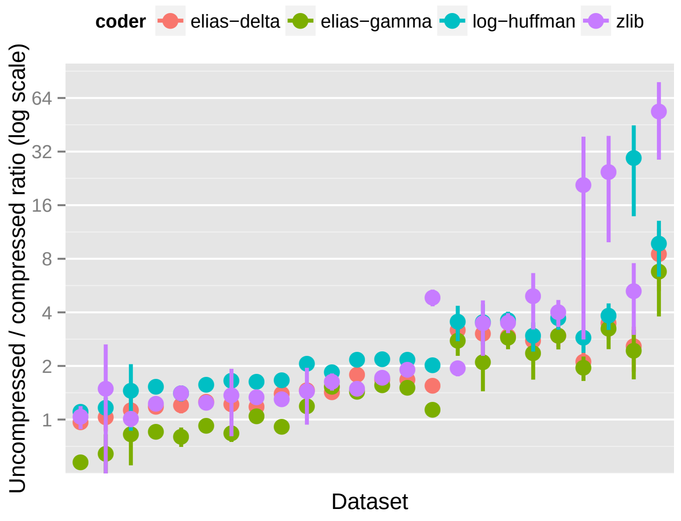
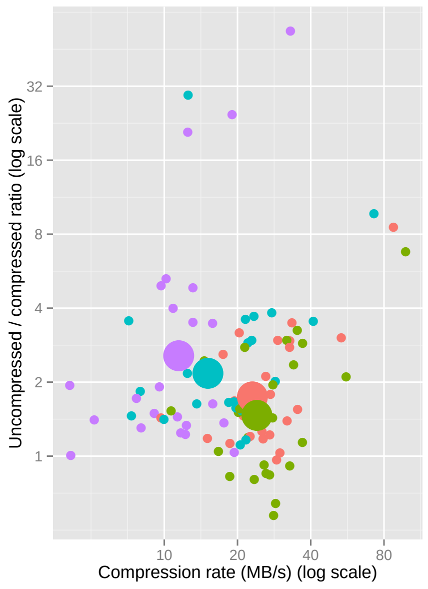


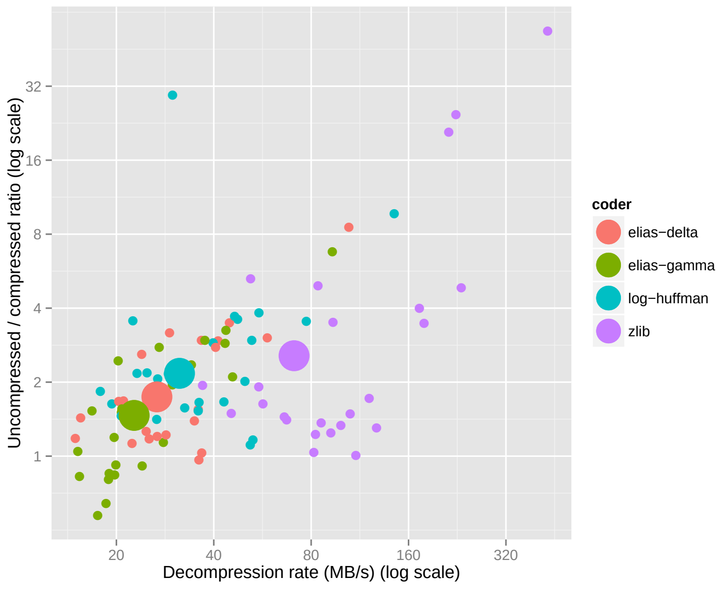
Figure 4: Encoded sizes broken down by dataset. Bars are 1/2 a standard deviation in height. The vertical axis is logarithmic, and datasets are sorted by compression ratio

Speed vs space: Space is often not the most important criterion for an encoder: in the context of a time series data store, encoding and decoding throughput and latency are critical. A more careful evaluation on portions of streams could separate throughput and latency; here we confound the two by measuring times on entire streams, but do separate encode and decode time. Figure 5 shows the result for all datasets. Since there are over many datasets, we also show the mean over all datasets for each encoder, which should not be granted too much weight (it is unclear whether the datasets are "equal").

A few preliminary findings stand out. First, performance for each encoder varies highly across datasets. On encoding, the Elias gamma and delta encoders are consistently faster than the other two techniques, and the Log-Huffman encoder is still typically faster than zlib. However, on decode, the reverse is true: zlib is consistently faster. As observed in Figure 4, Elias gamma coding is worst among the encoders in terms of space, and although Elias delta coding is occasionally in terms of compression ratio than log-huffman, the latter generally performs better (but on some datasets much worse than zlib).



(a) Encoding space vs speed



(b) Decoding space vs speed

Figure 5: Space vs speed comparison of encoders on all datasets, with a large dot as the mean of all datasets. The presentation is slightly unconventional; the "better" frontier is towards the right and top of the graph.

A brief excursion into a profiler revealed that significant time is spent in bitstream code. We have done some preliminary optimization while still maintaining the bitstream interface by providing an interface for writing multiple bits that shifts in blocks of bytes instead of individual bits[4]. However, this is not enough; parallel shifting and some of the avenues in Section 8 will likely be necessary.

Random access: Though we evaluate space and speed quantitatively here, recall that one of our goals was to support random access. The encoders vary in how easily this can be achieved (we have not implemented these techniques):

- Log-huffman: requires reading the dictionary (which must be less than 74 bytes, and is typically much smaller), and then access can be resumed from any point in the stream by knowing an index to bit offset mapping and the value at that index.
- Universal codes: only require an index to bit offset mapping and the value at that index.
- zlib: by default, there is no means for accessing arbitrary positions in the encoded stream. It is possible to provide access points by performing a "full flush" while encoding and inflate from those points, but doing so requires saving a block of uncompressed data equal to the compressor window size at each

[4]`https://github.com/mrcaps/generic-ts-encoding/blob/master/compressors/src/compressor/bitstream.hpp`

"sync point" (by default 32KB)[5] and reading it back in when decoding. Other solutions, such as RAzip[6], might be promising.

7 Limitations

The encoding techniques described above cannot handle missing data (e.g., `NaN` values), which were present in two of the datasets we analyzed. Since both these datasets had a fixed sample period, one solution would be to include explicit timestamps and skip timestamps with missing values. However, maintaining a fixed sample period is desirable, so approaches such as a separate bitmap index may be fruitful.

Our current procedure for finding the minimum exponent of a stream involves a full scan. This precludes streaming compression and can be cache-unfriendly. There is some low-hanging fruit for overcoming this limitation: most clearly, simply blocking the encoding. Also, it is likely often the case that the distribution can be estimated from the beginning of the stream. As a hint that this approach could bear fruit, Figure 6 shows an ECDF of the delta distribution of the entire stream (Prefix=999, in blue) as compared to the distributions of shorter prefixes (10, 20, ...). Even using 1-2% of this particular stream seems to produce a reasonable approximation of the complete distribution.

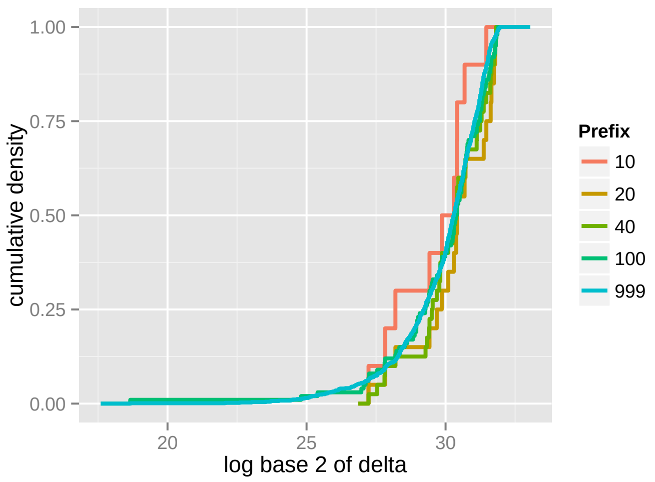


Figure 6: Closeness of prefix delta distribution to entire stream delta distribution

8 Related and future work

Here we touch on a few related areas of time series encoding, and how they might be useful for generic lossless numeric time series compression.

Lossy fixed-point compression: Doing justice to the vast collection of domain-specific time series compression techniques here would not be possible. Many of these are lossy, such as MP3 and Vorbis for audio, a variety of specialized techniques for ECG [9], and so forth.

Lossless fixed-point compression: There are a number of lossless fixed-point stream coders; familiar ones are used for audio signals. One prominent codec is FLAC [4]. Applying at least parts of the encoder to more general numeric time series data might be promising—in particular, its option to perform polynomial fitting and adaptive Rice coding of residuals. Similar techniques for fitting shapes to streams could exploit the reasonably general "smoothness" property even further.

[5]`http://svn.ghostscript.com/ghostscript/tags/zlib-1.2.3/examples/zran.c`
[6]`http://sourceforge.net/projects/razip/`

Lossless floating point compression: An interesting point of comparison would be a lossless floating-point compressor such as FPC [3], which for 64-bit floating-point data provides compression ratios similar to a variety of generic compressors but with higher throughput. However, it has been evaluated on data that with very limited compressibility; testing it using the data from the archive (and its significantly more-compressible data) would be intriguing. The authors find that gzip-style coders sometimes work surprisingly well, which is interesting in light of the results in Section 6.

Encoding benchmarks: More broadly, this work could be more useful for a different purpose: a start at a benchmark for testing generic lossless numeric time series encoders. To our knowledge, no such suite currently exists, and its existence is necessary and valuable for effective evaluation of encoders for a time series database.

Performance: In full hindsight, attempting to implement the entirety of a high-performance encoder that works on multiple data widths within the scope of this project was not the best idea. Many of the optimizations required to achieve high performance appear to be fairly close to the metal, and primarily involve reducing the overhead of bit-level operations. It seems we are not the first to observe that non-byte-aligned encoders are slow; for instance, Google's work on high throughput compression in Snappy and Gipfeli [12] does not use Huffman codes or arithmetic codes for performance reasons. Just as this project was coming to an end, we found some work using SIMD instructions to reduce this overhead for Elias gamma coding [17]. In a timely development, the AVX2 extensions that will appear in the Haswell microarchitecture in the near future will support vector shifts [8], likely allowing for even more potential for bit-level encoders.

References

[1] Daniel Abadi, Samuel Madden, and Miguel Ferreira. Integrating compression and execution in column-oriented database systems. In *SIGMOD*, 2006.

[2] Eric Anderson, Martin Arlitt, Charles B Morrey III, and Alistair Veitch. DataSeries: An efficient, flexible data format for structured serial data. Technical Report HPL-2009-323, HP Labs, 2009.

[3] M. Burtscher and P. Ratanaworabhan. FPC: A high-speed compressor for double-precision floating-point data. *IEEE Transactions on Computers*, 58(1):18–31, 2009.

[4] Josh Coalson. FLAC - free lossless audio codec. `http://flac.sourceforge.net`.

[5] Luca Deri, Simone Mainardi, and Francesco Fusco. tsdb: a compressed database for time series. In *TMA*, 2012.

[6] Google. Snappy: A fast compressor/decompressor. `https://code.google.com/p/snappy/`.

[7] HP. Vertica enterprise edition 6.0: Data encoding and compression. `https://my.vertica.com/docs/6.0.1/HTML/index.htm#2415.htm`.

[8] Intel Corporation. Intel advanced vector extensions programming reference, June 2011.

[9] S. M S Jalaleddine, C.G. Hutchens, R.D. Strattan, and W. Coberly. ECG data compression techniques-a unified approach. *IEEE Transactions on Biomedical Engineering*, 37(4):329–343, 1990.

[10] U. Kang, Hanghang Tong, Jimeng Sun, Ching-Yung Lin, and Christos Faloutsos. Gbase: an efficient analysis platform for large graphs. *The VLDB Journal*, 21(5), 2012.

[11] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandier, Lyric Doshi, and Chuck Bear. The Vertica analytic database: C-store 7 years later. *VLDB Endowment*, 5(12), 2012.

[12] R. Lenhardt and J. Alakuijala. Gipfeli - high speed compression algorithm. In *Data Compression Conference (DCC)*, 2012.

[13] Charles Loboz, S. Smyl, and S. Nath. DataGarage: Warehousing Massive Performance Data on Commodity Servers. In *VLDB*, 2010.

[14] Sergey Melnik, Andrey Gubarev, J.J. Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. Dremel: interactive analysis of web-scale datasets. In *VLDB*, 2010.

[15] Jonathan Moore. Saturnalia DB, March 2013. `http://saturnaliadb.org/`.

[16] Tobias Oetiker. RRDtool, 2011. `http://www.mrtg.org/rrdtool/`.

[17] Benjamin Schlegel, Rainer Gemulla, and Wolfgang Lehner. Fast integer compression using simd instructions. In *Workshop on Data Management on New Hardware (DaMoN)*, 2010.

[18] Jin Shieh and Eamonn Keogh. UCR Archive. `http://www.cs.ucr.edu/~eamonn/iSAX/iSAX.html`, `http://www.cs.ucr.edu/~eamonn/Keogh_Time_Series_CDrom.zip`.

[19] Benoit Sigoure. OpenTSDB. `http://opentsdb.net/`.

[20] RS Weigel, DM Lindholm, Anne Wilson, and Jeremy Faden. TSDS: high-performance merge, subset, and filter software for time series-like data. *Earth Science Informatics*, 3(1-2), June 2010.