An Evaluation of Methods of Compressing Doubles

Jacob Spiegel, University of Chicago

ABSTRACT

Data compression is a problem with far-reaching implications across science and industry. In the era of big data, methods for efficient compression are crucial to achieve compact data representation, low-latency data transfers, and high-throughput during query execution. Due to the explosion of Internet-of-Things applications, a large portion of this data is in the form of double-precision floating-point numbers. Despite the plethora of methods for compression, a comprehensive evaluation across real-world data and applications is still missing. In this paper, we perform such a comparison of methods and evaluate their performance in terms of compression ratio and throughput achieved across two dataset repositories of time series and featurized machine-learning problems, as well as on a dataset of machine logs.

ACM Reference Format:
Jacob Spiegel, University of Chicago. 2020. An Evaluation of Methods of Compressing Doubles. In *2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, Portland, OR, USA, 3 pages. https://doi.org/10.1145/3318464.3384415

1 INTRODUCTION

Compression mechanisms have become increasingly important in the era of big data. Despite a steady decrease in storage cost, the amount of data collected is increasing exponentially. This data is generated by a plethora of sensors, mobile devices, and machine logs, and a large portion of them are in the form of double-precision floating-point numbers.

However, only a few methods exist for general-purpose compression of doubles - methods of compressing integers are more common. In their corresponding papers, existing methods have performed evaluations in a variety of programming languages and on machines with vastly different configurations. In addition, different metrics were used to determine their effectiveness, and the evaluations were on significantly different datasets, resulting in difficulty in understanding their relative performances.

SIGMOD '20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06.
https://doi.org/10.1145/3318464.3384415

Contribution: The goal of this paper is to address the aforementioned problem by (i) implementing some of today's most promising compression methods for doubles in the same programming language; and (ii) evaluating these methods across real-world datasets using the same machine and the same metrics. The objective is to identify the state-of-the-art method for compressing doubles.

Compression Methods: To achieve this goal, I first review two recent methods:

- **Gorilla [5]:** Originally designed by Facebook for compressing machine logs, which were most often doubles with a fractional portion of 0. Each double is xor-ed with the previous double, and that result is stored as either a single zero bit if the xor is zero or as its number of leading zeroes, number of meaningful bits, and the meaningful bits themselves if the xor is not zero.
- **Sprintz [1]:** Originally developed by researchers at MIT for the compression of integer-valued time series. Each integer is subtracted by the previous integer, resulting in error e. For each block of n errors, the min of the number of leading zeroes (m) is found, and each error in that block is stored as $16 - m$ bits, along with m itself. A block is considered a zero-block if $m = 16$ for that block, and a run of zero blocks is compressed as a single zero followed by the number of zero blocks in the run.

I implemented these compression methods in Java[1]. To make Sprintz work for doubles I changed the bitwidth from 16 to 64 bits. In addition, I extended Sprintz by calculating the error as $x_i \oplus x_{i-1}$ instead of $x_i - x_{i-1}$, which significantly increased (approximately by 2x, compared to the baseline) the amount of compression obtained with no impact on throughput. For baselines, I use two approaches. First, I consider the compression achieved by converting the datasets directly to their binary representations. This baseline serves as an "upper bound" on the performance of the compression methods; they should all perform at least as well as this baseline and any additional compression beyond that point is what is of interest. Second, I consider the compression achieved by gzip, a widely used compression software that works on any kind of data. This serves as a "target" for the double-specific methods; all competing methods should ideally achieve compression ratios near, or even lower than, gzip.

[1]The source code is publicly available at https://github.com/UCHI-DB/DoubleCompression

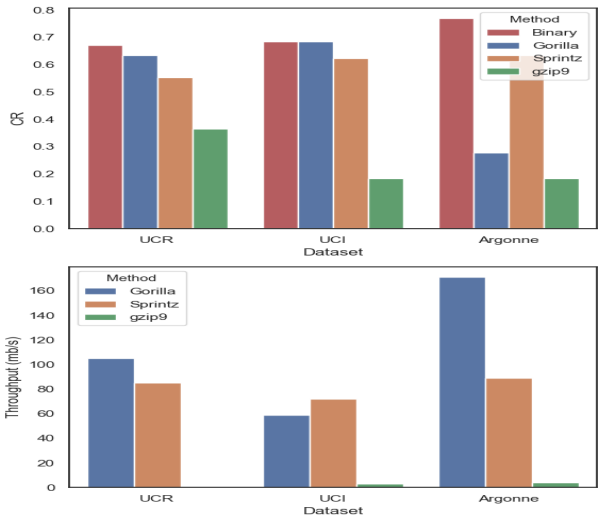

Figure 1: Comparison of compression methods with (a) showing the compression ratio on each dataset (a lower compression ratio is better, with 1 indicating no compression and 0 indicating a theoretical "full compression" (an unreachable lower bound in which the compressed data has a size of zero bytes); and (b) showing the throughput on each dataset measured in mb/s (the binary compression, which has an average throughput of 488 mb/s, has been omitted for the purpose of keeping gzip9 visible).

2 DATASETS

I evaluate these methods on two standard dataset repositories with numeric data: namely, the UCR Time-Series Archive [2] and a sample of the UCI Machine-Learning Datasets Archive [3]. The UCR repository contains 128 datasets, and from the UCI repository we sampled 121 machine learning datasets and used standard featurization methodologies to convert non-numeric fields into doubles. In addition, I evaluate the methods on a dataset of machine logs collected from Argonne National Labs, which I analyze separately, as over 80% of the doubles in this dataset had a fractional portion of 0, making them more similar to integers.

3 RESULTS

As seen in Figure 1, both Gorilla and Sprintz are relatively poor compressors compared to gzip on UCR and UCI. Gorilla is particularly bad, compressing an average of only 1% of the amount that gzip does (compared to a baseline of the binary compression), while Sprintz compresses an average of 20% of the amount that gzip does. The fact that neither method comes close to gzip's compression indicates that there is much room to improve upon our current double compression methods. However, on the Argonne dataset, Gorilla performs very well, achieving an average compression ratio of 0.28, compared to gzip's 0.18. This is due to the aforementioned fact that the Argonne dataset is largely composed of doubles with a fractional portion of 0, which is the kind of data Gorilla was originally developed to work best on.

Components	CR
Binary	0.67
gzip	0.39
Xor	0.71
Null Suppression	0.71
Bit Packing	0.7
Xor + NS	0.65
Xor + BP	0.54
NS + BP	0.7
All	0.54

Table 1: The decomposed forms of Sprintz, along with binary and gzip compression, evaluated on UCR.

I also evaluate Sprintz's internal components in order to better understand which component is responsible for the achieved compression (see Table 1). I found that the compression is the result of the combination of calculating the error as the xor with the previous double and the bit packing that the algorithm performs. This implies that Sprintz can be further improved by reducing the error calculated, likely by devising a better prediction method. FCM and DFCM [4], which attempt to predict values based on previously encountered patterns, are logical algorithm choices for prediction, but unfortunately in practice these provided no additional compression based on my evaluation. One method that could prove promising is the FIRE forecaster described in the Sprintz paper, which does not work as-is on non-integer data, but may be able to be adapted to work on doubles.

The advantage that Gorilla and Sprintz have, regardless of dataset, is their speed. gzip is notably slow, compressing at an average rate of 1.5 mb/s, whereas Gorilla and Sprintz compress at an average rate of 102 mb/s and 83 mb/s, respectively. This makes Gorilla and Sprintz 68 and 55 times faster than gzip, respectively. Additionally, while all efforts were made to avoid the slowdowns associated with Java, these algorithms would still likely have an even higher throughput if implemented in another programming language.

From these results, we see that there is a trade-off between speed and compression when picking between gzip and a double-specific compressor, and the user must choose which to use based on their needs. Further research will hopefully reveal correlations between dataset metrics and performance of the algorithms, allowing low-cost analysis of the dataset to help guide this decision.

REFERENCES

[1] Davis Blalock, Samuel Madden, and John Guttag. 2018. Sprintz: Time Series Compression for the Internet of Things. *arXiv preprint arXiv:1808.02515* (2018).

[2] Hoang Anh Dau, Eamonn Keogh, Kaveh Kamgar, Chin-Chia Michael Yeh, Yan Zhu, Shaghayegh Gharghabi, Chotirat Ann Ratanamahatana, Yanping, Bing Hu, Nurjahan Begum, Anthony Bagnall, Abdullah Mueen, Gustavo Batista, and Hexagon-ML. 2018. The UCR Time Series Classification Archive. https://www.cs.ucr.edu/~eamonn/time_series_data_2018/.

[3] Dheeru Dua and Casey Graff. 2017. UCI Machine Learning Repository. http://archive.ics.uci.edu/ml

[4] B. Goeman, H. Vandierendonck, and K. de Bosschere. 2001. Differential FCM: increasing value prediction accuracy by improving table usage efficiency. In *Proceedings HPCA Seventh International Symposium on High-Performance Computer Architecture*. 207–216. https://doi.org/10.1109/HPCA.2001.903264

[5] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. 2015. Gorilla: A Fast, Scalable, In-memory Time Series Database. *Proc. VLDB Endow.* 8, 12 (aug 2015), 1816–1827. https://doi.org/10.14778/2824032.2824078