How to Make your Duck Fly: Advanced Floating Point Compression to the Rescue

Panagiotis Liakos
Athens University of Economics and Business
Athens, Greece
panagiotisliakos@aueb.gr

Katia Papakonstantinopoulou
Athens University of Economics and Business
Athens, Greece
katia@aueb.gr

Thijs Bruineman
DuckDB Labs
Amsterdam, The Netherlands
thijs@duckdblabs.com

Mark Raasveldt
DuckDB Labs
Amsterdam, The Netherlands
mark@duckdblabs.com

Yannis Kotidis
Athens University of Economics and Business
Athens, Greece
kotidis@aueb.gr

ABSTRACT

The massive volumes of data generated in diverse domains, such as scientific computing, finance and environmental monitoring, hinder our ability to perform multidimensional analysis at high speeds and also yield significant storage and egress costs. Applying compression algorithms to reduce these costs is particularly suitable for column-oriented DBMSs, as the values of individual columns are usually similar and thus, allow for effective compression. However, this has not been the case for binary floating point numbers, as the space savings achieved by respective compression algorithms are usually very modest. We present here two lossless compression algorithms for floating point data, termed Chimp and Patas, that attain impressive compression ratios and greatly outperform state-of-the-art approaches. We focus on how these two algorithms impact the performance of DuckDB, a purpose-built embeddable database for interactive analytics. Our demonstration will showcase how our novel compression approaches *a*) reduce storage requirements, and *b*) improve the time needed to load and query data using DuckDB.

1 INTRODUCTION

Data management systems have traditionally employed data compression techniques for many decades now, as an effective means for reducing storage space, network bandwidth and egress costs [2]. In addition to more efficient disk and network I/O, the merits of compression can also be observed with regards to the overall database performance. As the number of records that fit into the buffer space grows, so does the buffer hit ratio [2]. Moreover, the increased number of records that a page can accommodate after compression allow for more effective clustering of records used together [1].

Applying compression is particularly suitable for column stores, that store the data of each column individually, and thus, can easily employ different encoding techniques, depending on the attribute types and value distributions. However, even though several techniques allow for compressing text or integer numbers by a factor of two or more, compressing floating point numbers has not been equivalently effective. Available techniques offer very modest savings, and general purpose compression algorithms are very slow.

In this work, we present two algorithms for compressing floating point data, termed Chimp and Patas, respectively. Chimp [4] is a novel lossless streaming compression algorithm, providing significant space savings that greatly outperform the current state-of-the-art streaming approaches, while also being competitive with much slower, yet extremely effective general purpose compression schemes, as well as lossy approaches [3]. Chimp performs a bitwise XOR operation between the current value and a previous value to come up with a resulting set of bits that is likely to contain a lot of zeros, as neighboring data points do not change significantly. The choice of the previous value to be used is based on the number of similar trailing bits with the current value, in an effort to maximize the number of trailing zeros in the resulting XORed value. We use two control bits for each data point, to differentiate between a total of four possible encoding formats, that enable us to minimize the bits needed to represent each point. Moreover, due to the use of a sophisticated data structure that enables us to quickly retrieve the best candidate values, Chimp preserves the compression and decompression speed of earlier streaming approaches.

Patas is a variation of Chimp, that focuses on providing decompression speed that is comparable with that of reading uncompressed data. To this end, Patas applies zero-padding to the meaningful bits of each resulting XOR-ed value, to form 1-byte aligned representations and avoid having to perform any bit-level processing, as Chimp does. Naturally, these design choices have an effect on the space savings of Patas, as there exists a trade-off between the compression ratio we can achieve and the decompression speed we may offer. Still, the space efficiency of Patas is significantly better than what earlier streaming approaches offer. Both Chimp and Patas are part of the embeddable DuckDB [9] database for interactive analytics, as of its 0.6.0 release. The two algorithms are used for the compression of double precision floating point numbers, offering significant improvements in the overall performance of the database.

For our demonstration, we have designed a user-friendly web application that enables users to witness the impressive space savings and performance benefits that Chimp and Patas provide, as well as gain insights with regards to the added value that compression algorithms may offer to an established and widely used database tool. The underlying system features three identical setups of DuckDB deployments that operate in three different modes

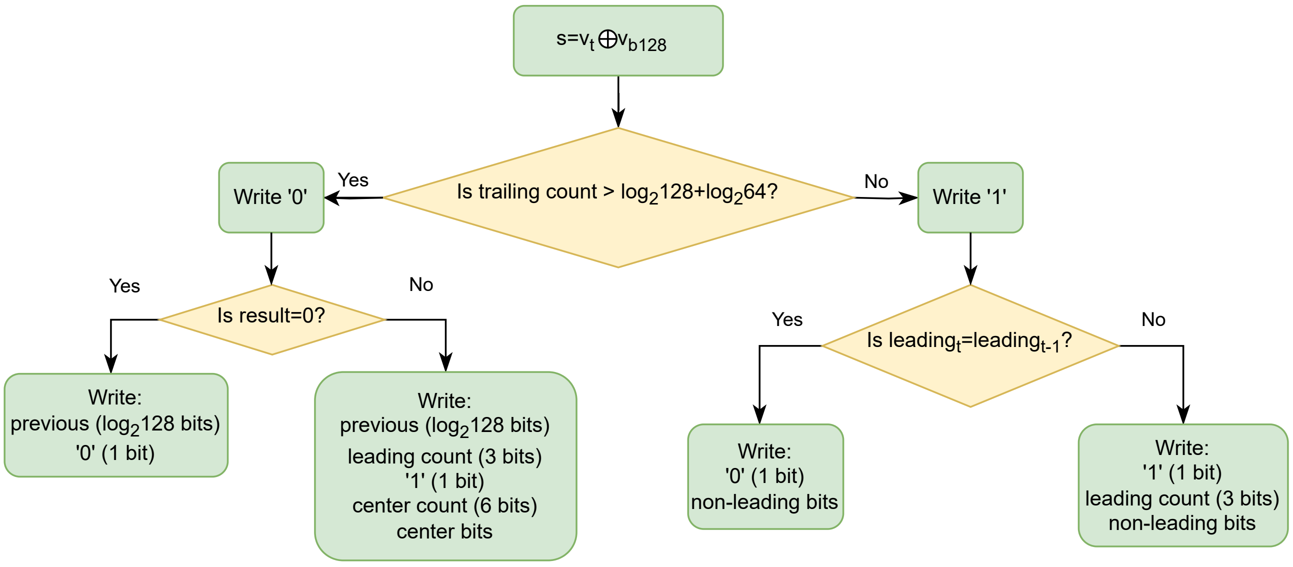


Figure 1: Chimp compression algorithm.

regarding floating point data: a) uncompressed, b) compressed with Chimp, and c) compressed with Patas. The interface of the web application comprises two main sections that the user can interact with. The first, focuses on loading data to the database and enables the user to inspect a number of available files with raw data and submit one or more of them to be loaded in the database. The second, provides a number of different predefined queries and enables the user to submit them for execution. More importantly, we provide an SQL editor with syntax highlighting and offer more experienced users with the opportunity to submit custom queries to freely explore the performance of the database under the three different modes of operation. The results of each load or query action are visualized using intuitive animated charts that are updated after the completion of each request.

A full description of the Chimp algorithm along with extensive experiments over a wide range of large real-world datasets can be found in [4]. Here, we focus on the key points of Chimp, its extension termed Patas, and the proposed demonstration scenarios.

2 CHIMP

In this section, we provide the details of the Chimp compression algorithm [4], focusing on the Chimp$_{128}$ variant that exploits the best of 128 previously encountered values. In what follows we will use the term Chimp to refer to this variant.

2.1 Bitwise XOR operation

Similarly to earlier approaches [8], Chimp uses a bitwise XOR operation to exploit similarities between consecutive measurements. More specifically, the intent of this operation is to generate a resulting value with long runs of leading and trailing zeros. The bitwise XOR operation in Chimp is performed between the current value v_t and the best of 128 previous values in terms of most trailing zeros, $v_{b_{128}}$, as shown in Figure 1. If the resulting number of trailing zeros *surpasses* the number of bits needed to denote the previous value used ($log_2 128$ bits) plus the number of bits required to specify the number of meaningful bits ($log_2 64$ bits), then we make use of and actually store the previous value used (two bottom-left cases of Figure 1). Otherwise, the use of $v_{b_{128}}$ is not particularly useful and, due to the flexibility of Chimp, we can use the immediately previous value v_{t-1} instead, and avoid wasting additional bits to denote the previous value used (two bottom-right cases of Figure 1).

2.2 Leading Zero Bits

Earlier approaches [8] use 5 bits to denote up to 31 leading zeros in the resulting XORed value. We initially considered the use of 4 bits at 2-bit granularity to represent the length of leading zero bits through the mapping $f : A \longrightarrow B$, where $A = \{0, 1, \ldots, 2^n - 1\}$, $B = \{0, 1, \ldots, 2^{n-1} - 1\}$, and $n \in \mathbb{N}$, given by

$$f(x) = \begin{cases} \frac{x}{2}, & \text{when x is even, and} \\ \frac{x-1}{2}, & \text{when x is odd} \end{cases} \quad (1)$$

to encode the run of leading zeros using one fewer bit. In addition, we used mapping $f' : B \longrightarrow C$, where $C = \{0, 2, \ldots, 2^n - 2\}$, given by $f'(x) = 2x$, to decode the run of leading zeros. It is evident that the above mappings are lossy in the case of odd numbers, with an error that is always equal to 1. Therefore, to preserve all information for odd numbers, we have to encode a zero bit along with the remaining XORed value. Despite this fact, this representation is always at least as compact as that of earlier approaches that always use 5 bits [4].

Moreover, Chimp further improves the space efficiency of the leading zeros representation by exploiting their distribution, and especially the fact that small values of runs are rarely encountered [4]. Thus, the final design for Chimp uses only 3 bits to represent up to 24 leading zeros with an *exponentially decaying step* between the mapped values. The actual steps used are 0, 8, 12, 16, 18, 20, 22, 24 and provide Chimp with improved compression gains compared to the 4-bit representation with 2-bit granularity for a wide range of datasets.

Additionally, there exist cases in which Chimp capitalizes on similarities with regard to the number of leading zeros in consecutive XORed values to induce further savings. Chimp considers two cases of XORed values whose number of trailing zeros is less than a certain threshold. When a value's number of leading zeros is *exactly equal* to the previous value's number of leading zeros, we can simply write a single '0' control bit, instead of spending 3 bits to specify this number again. Otherwise, we write control bit '1' as well as the number of leading zeros. We note here that using our 3-bit representation of leading zero bits, different numbers are mapped to the same value, e.g., 16 and 17 are both mapped to 100 and are thus, considered equal. This favors the first case of our strategy, and therefore, our overall approach becomes very effective in terms of exploiting leading zero bits.

2.3 Detailed Chimp Compression

Chimp performs a bitwise XOR between the current value and the best of 128 previous values in terms of most trailing zeros, $v_{b_{128}}$, and encodes the result with the following variable length encoding scheme.

- The first value is stored with no compression.
- When XOR with the previous value has more than $log_2 128+ log_2 64$ trailing zeros, we store a single '0' bit followed by a $log_2 128$-bit index denoting the previous values used and either:
  - Control bit '0': If the result is zero, i.e., the values are identical.
  - Control bit '1': If the result is not zero, we store the length of the number of leading zeros in the next 3 bits, then store the length of the meaningful XORed value in the next 6 bits. Finally store the meaningful bits of the XORed value.
- When XOR has $log_2 128 + log_2 64$ or less trailing zeros, we store a single '1' bit followed by either:
  - Control bit '0': If the number of leading zeros is exactly equal to the previous leading zeros, we use that information and just store the meaningful XORed value.
  - Control bit '1': We store the length of the number of leading zeros in the next 3 bits and the meaningful bits of the XORed value.

This operation is portrayed with the diagram of Figure 1.

3 PATAS

The goal of Chimp is to maximize the space savings for the representation of floating point numbers while also preserving the compression and decompression speed of earlier approaches. The Patas variant we discuss here, aims at providing decompression rates that are comparable with those of reading uncompressed data, at the cost of slightly smaller space savings.

Chimp can exploit various patterns found in streams of floating point values using the different control bits. However, not all of these patterns are present in all data sets. The control bits also come at a performance cost –as the decompression routine requires branching and has data dependencies that prevent efficient pipelined execution on modern CPUs. Patas offers faster decompression by specializing on a single pattern instead –namely when the control bits are '01'. By specializing on a single pattern, branching and data dependencies during decompression are avoided, and the decompression can be performed significantly faster. In addition, Patas performs byte-aligning of values instead of bit-aligning to further speed up decompression.

Specializing on this single pattern results in only a slightly reduced compression ratio when this pattern is the most common one in a given data set. When the other patterns occur more frequently, the system can detect this and automatically fallback to using Chimp to take advantage of the more flexible approach taken by the Chimp algorithm.

Experimental results show that decompressing values using the original Chimp algorithm is 5x slower than reading uncompressed values.[1] The Patas variation can decompress values in a fraction of the time that Chimp needs, taking only twice as much time as reading uncompressed values would require.[2] Thus, Patas provides faster decompression at the cost of slightly reduced compression ratio.

Depending on the application at hand, the user may opt to use Chimp to achieve the maximum space savings, Patas to get more efficient decompression with a smaller compression ratio, or store the data in uncompressed format, to achieve ultimate decompression speed at the cost of acquiring no space savings at all.

4 DEMONSTRATION SCENARIOS

Our demonstration is based on a web application with an intuitive and user friendly interface that guides users to explore the performance of a popular database under different modes of operation regarding the representation of floating point values. More specifically, the web interface allows for submitting HTTP requests to three web applications running in an identical containerized environment, that use an embedded DuckDB database [9], configured to apply *a*) no compression, *b*) Chimp compression, and *c*) Patas compression, respectively. The configuration commands used are shown in Figure 2. In this way, we enable users to measure the impact of using compression while also answering the question why Chimp and Patas are excellent choices for handling massive volumes of floating point data.

```
PRAGMA force_compression=uncompressed

PRAGMA force_compression=chimp

PRAGMA force_compression=patas
```

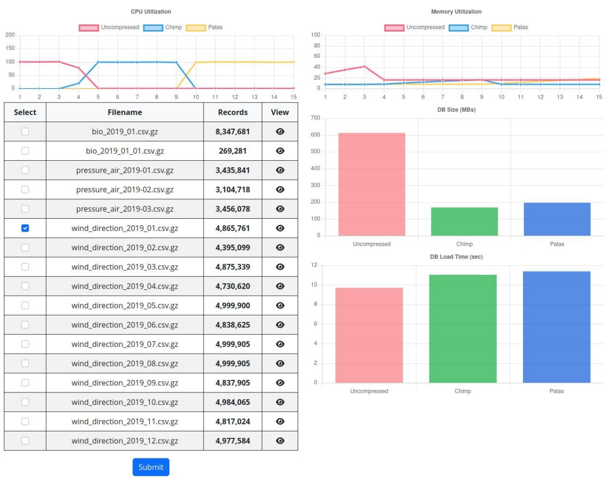
Figure 2: DuckDB compression configuration commands for floating point data.

Our demonstration comprises two parts: First, we focus on loading raw data to the database and measure the respective storage requirements and time needed. Second, we allow users to issue queries, either by submitting one of the many available predefined queries that investigate different aspects of query processing, or by freely composing queries of their own. In both cases, the interface offers real-time monitoring of CPU and memory utilization. We use a wide range of large scale real-world open-access datasets extracted from the NEON repository [5–7], and use the latest release of DuckDB, i.e., 0.6.1, that features implementations of both Chimp and Patas algorithms.

4.1 Loading Data

In the first part of our demonstration, the user is presented with the list of raw data files of Figure 3, that shows the name of the file, and the number of available records. The user may view the first lines of each file to gain insights on its attributes by clicking a button next to the file name. Having viewed the information available, the user may select one or more files and

[1] https://github.com/duckdb/duckdb/pull/4878
[2] https://github.com/duckdb/duckdb/pull/5044



Select	Filename	Records	View
☐	bio_2019_01.csv.gz	8,347,681	👁
☐	bio_2019_01_01.csv.gz	269,281	👁
☐	pressure_air_2019-01.csv.gz	3,435,841	👁
☐	pressure_air_2019-02.csv.gz	3,104,718	👁
☐	pressure_air_2019-03.csv.gz	3,456,078	👁
☑	wind_direction_2019_01.csv.gz	4,865,761	👁
☐	wind_direction_2019_02.csv.gz	4,395,099	👁
☐	wind_direction_2019_03.csv.gz	4,875,339	👁
☐	wind_direction_2019_04.csv.gz	4,730,620	👁
☐	wind_direction_2019_05.csv.gz	4,999,900	👁
☐	wind_direction_2019_06.csv.gz	4,838,625	👁
☐	wind_direction_2019_07.csv.gz	4,999,905	👁
☐	wind_direction_2019_08.csv.gz	4,999,905	👁
☐	wind_direction_2019_09.csv.gz	4,837,905	👁
☐	wind_direction_2019_10.csv.gz	4,984,065	👁
☐	wind_direction_2019_11.csv.gz	4,817,024	👁
☐	wind_direction_2019_12.csv.gz	4,977,584	👁

Figure 3: Loading data example. The user can select datasets to be loaded and monitor the CPU and memory utilization as well as the loading time required for each compression mode.

submit them for insertion in the database. Then, the web application sequentially executes the insertion in each of the three database instances. If the user has selected more than one files of the same dataset, the first file will loaded with a 'CREATE TABLE' command, whereas a 'COPY' command will be used for the rest of the files, as shown in Figure 4. After the execution of these commands in each of the three databases, the user is shown the size of the database and the time needed to complete the loading, as we see in Figure 3.

```
CREATE TABLE table_name AS SELECT *
FROM read_csv('filename.csv', AUTO_DETECT=TRUE)

COPY table_name
FROM 'filename.csv' ( DELIMITER ',', HEADER )"
```

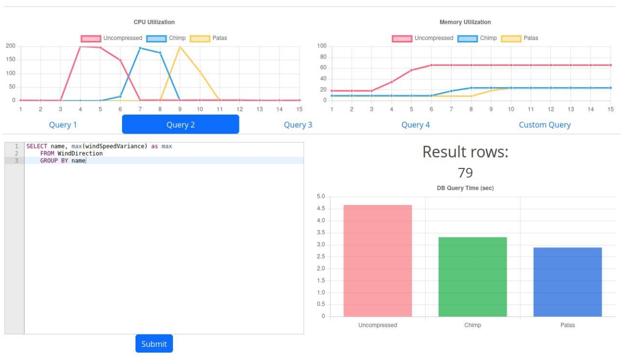
Figure 4: DuckDB loading commands used for creating tables and loading data from files.

Through this first part of the demonstration, both inexperienced users and experts in data management, will gain insights on the space savings the state-of-the-art floating point compression approaches offer and the impact these savings have on the database performance during data ingestion.

4.2 Querying Data

In the second part of our demonstration, the user is given the opportunity to query the three database instances. As we see in Figure 5, the user may submit one of the many available predefined queries offered or submit a query composed on the fly.

The first option is addressed to users that quickly want to grasp the advantages of compression with regard to query processing, through a series of carefully selected queries. The queries include *a*) ungrouped aggregates, *b*) grouped aggregates with different levels of cardinality, *c*) grouped aggregates with a filter, and *d*) joins. The different parameters involved in the queries are randomly updated as the user navigates in the web application and submits queries for execution, so that we limit the chance that the databases serve cached results.



```
SELECT name, max(windSpeedVariance) as max
    FROM WindDirection
    GROUP BY name
```

Result rows:
79

Figure 5: Querying data example. The user can execute arbitrary queries and monitor CPU and memory utlization as well as query execution time for each compression mode.

The second option, that allows for composing arbitrary queries, is addressed to more experienced users that want to submit additional queries to the three database instances, to further evaluate their performance. The users are provided with a rich SQL editor with syntax highlighting and any syntax errors are reported to help users compose valid queries.

While a query is executed, the user can monitor its impact in terms of resource consumption, and after the execution the user is presented with the number of rows in the result and the time needed to process the query, as we see in Figure 5. Through this second part of the demonstration, the users will obtain a clear understanding of the impact of compression in query processing, and the merits associated with the improved I/O and better utilization of memory.

ACKNOWLEDGMENTS

Panagiotis Liakos has received funding from the Hellenic Foundation for Research and Innovation (HFRI) and the General Secretariat for Research and Technology (GSRT), under grant agreement No 779. Yannis Kotidis has been supported by the European Union under Horizon Europe project EVENFLOW agreement No 101070430.

REFERENCES

[1] P.A. Alsberg. 1975. Space and time savings through large data base compression and dynamic restructuring. *Proc. IEEE* 63, 8 (1975), 1114–1122.
[2] G. Graefe and L.D. Shapiro. 1991. Data compression and database performance. In *[Proceedings] 1991 Symposium on Applied Computing*. 22–27.
[3] Xenophon Kitsios, Panagiotis Liakos, Katia Papakonstantinopoulou, and Yannis Kotidis. 2023. Sim-Piece: Highly Accurate Piecewise Linear Approximation through Similar Segment Merging. *Proc. VLDB Endow.* 16, 8 (2023), 1910–1922.
[4] Panagiotis Liakos, Katia Papakonstantinopoulou, and Yannis Kotidis. 2022. Chimp: Efficient Lossless Floating Point Compression for Time Series Databases. *Proc. VLDB Endow.* 15, 11 (2022), 3058–3070.
[5] National Ecological Observatory Network (NEON). 2021. 2D wind speed and direction (DP1.00001.001). https://doi.org/10.48443/S9YA-ZC81
[6] National Ecological Observatory Network (NEON). 2021. Barometric pressure (DP1.00004.001). https://doi.org/10.48443/RXR7-PP32
[7] National Ecological Observatory Network (NEON). 2021. IR biological temperature (DP1.00005.001). https://doi.org/10.48443/JNWY-B177
[8] Tuomas Pelkonen, Scott Franklin, Paul Cavallaro, Qi Huang, Justin Meza, Justin Teller, and Kaushik Veeraraghavan. 2015. Gorilla: A Fast, Scalable, In-Memory Time Series Database. *Proc. VLDB Endow.* 8, 12 (2015), 1816–1827.
[9] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proc. of the 2019 Int. Conf. on Management of Data, SIGMOD Conference 2019, Amsterdam, The Netherlands, June 30 - July 5, 2019.* 1981–1984.