Low Level Big Data Compression

Jaime Salvador-Meneses[1], Zoila Ruiz-Chavez[1] and Jose Garcia-Rodriguez[2]

[1] *Universidad Central del Ecuador, Ciudadela Universitaria, Quito, Ecuador*

[2] *Universidad de Alicante, Ap. 99. 03080, Alicante, Spain*

Keywords: Big Data, Data Compression, Categorical Data, Encoding.

Abstract: In the last years, some specialized algorithms have been developed to work with categorical information, however the performance of these algorithms has two important factors to consider: the processing technique (algorithm) and the representation of information used. Many of the machine learning algorithms depend on whether the information is stored in memory, local or distributed, prior to processing. Many of the current compression techniques do not achieve an adequate balance between the compression ratio and the decompression speed. In this work we propose a mechanism for storing and processing categorical information by compression at the bit level, the method proposes a compression and decompression by blocks, with which the process of compressed information resembles the process of the original information. The proposed method allows to keep the compressed data in memory, which drastically reduces the memory consumption. The experimental results obtained show a high compression ratio, while the block decompression is very efficient. Both factors contribute to build a system with good performance.

1 INTRODUCTION

Machine Learning (ML) algorithms work iteratively on large datasets using read-only operations. To get better performance in the process, it's important to keep all the data in local or distributed memory (Elgohary et al., 2017), these methods base their operation on classical techniques that become complex when the amount of data increases considerably (Romangonzalez, 2012).

The reduction of the execution time is an important factor to work with Big Data which demands a high consumption of memory and CPU resources (Hashem et al., 2016). Some ML algorithms have been adapted to work with categorical data. This is the case of fuzzy-kMeans that was adapted to fuzzy-kModes (Huang and Ng, 1999) to work with categorical data (Gan et al., 2009).

Nowadays, it is a challenge to process data sets with a high dimensionality such as the census carried out in different countries (Rai and Singh, 2010). A census is a particularly relevant process and currently constitutes a fundamental source of information for a country (Bruni, 2004).

This work proposes a new method to represent and store categorical data through the use of bitwise operations. Each attribute f (column) is represented by a one-dimensional vector.

The method proposes to compress the information (data) into packets with a fixed size (16, 32, 64 bits), in each packet a certain amount of values (data) is stored through bitwise operations. Bitwise operations are widely used because they allow to replace arithmetics operations with more efficient operations (Seshadri et al., 2015). The validity of the method has been tested on a public dataset with good results.

This document is organized as follows: Section 2 summarizes the representation of the information and the categorical data compression algorithms, in Section 3 an alternative is presented for the representation of information by compression using bitwise operations, Section 4 presents several results obtained using the proposed compression method and, finally, Section 5 summarizes some conclusions.

2 COMPRESSION ALGORITHMS

The main goal of this work is the compression of categorical data, so in this section we present a summary of some methods for the compression of categorical data.

Categorical data is stored in one-dimensional vector or in matrix composed by vectors depending on

Salvador-Meneses, J., Ruiz-Chavez, Z. and Garcia-Rodriguez, J.
Low Level Big Data Compression.
DOI: 10.5220/0007228003530358
In *Proceedings of the 10th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K 2018) - Volume 1: KDIR*, pages 353-358
ISBN: 978-989-758-330-8

the information type. Matrix representation uses the vectorial representation of its rows or columns so it is useful to describe the storage using rows and columns.

The following describes some options of compression and representation of one-dimensional data sets (vectors).

Run-length Encoding: Consecutive sequences of data with the same value are stored as a pair $(count, value)$ in which *value* represents the value to be represented and *count* represents the number of occurrences of the value within the sequence.

There are variations to this type of representation in which if the sequence of equal values are repeated in different position of the vector, the value is stored and additional to this, the beginning and the total number of elements in each sequence are represented (Elgohary et al., 2016).

Offset-list Encoding: For each distinct value within the data set a new list is generated which contains the indexes in which the aforementioned value appears. In the case that there are two correlated set, a (x, y) pair is generated and the index in which the data pair appears is stored in the new list.

Figure 1 shows Run-Length encodig (RLE) and Offset-list encoding (OLE) compression schemas.

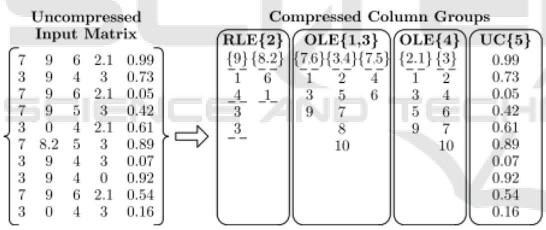

Figure 1: Compression examples (Elgohary et al., 2017).

GZIP: Compression is based in the DEFLATE algorithm[1] that consists in two parts: Lz77 and Huffman coding. The Lz77 algorithm compress the data removing redundant parts and the Huffman coding codes the result generated by Lz77 (Ouyang et al., 2010).

The classical compression methods, such as GZIP, considerably overloads the CPU which minimizes the performance gained by reducing the read/write operations, this fact makes them unfeasible options to be implemented in databases (Chen et al., 2001).

Bit Level Compression: REDATAM software[2] uses a distinct data compression schema that is based in 4-bytes blocks. Each block stores one or more values depending of the maximum size in bits required to store the values (De Grande, 2016).

[1] https://tools.ietf.org/html/rfc1952

[2] http://www.redatam.org

This compression format represents the most viable option when working with categorical data, because in most cases the information to be represented has a low number of different categories. This method uses the total amount of available bits in each block, so that a value can be contained in two different blocks of compressed data. Figure 2 shows the above.

Figure 2: REDATAM compression.

3 COMPRESSION APPROACH

In this section we propose a new mechanism for compressing categorical data, the compression method proposal corresponds to a variation of the **bit level compression** method described in Section 2. This method doesn't use all available bits because 32 may not be a multiple of the number of bits needed to represent the categories.

The numerical information of categorical variables is represented, traditionally, as signed integer values of 32, 16 or 8 bits (4, 2, 1 bytes). This implies that to store a numerical value it is necessary to use 32 bits (or its equivalent in 2 or 1 byte). We will consider the case in which the information is represented as a set of 4-bytes integer values.

Figure 3 represents the bit distribution of a integer value composed of 4 bytes.

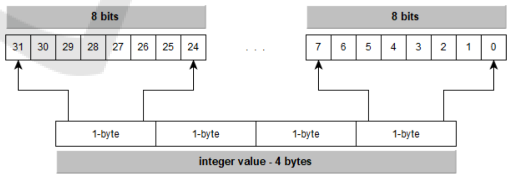

Figure 3: Representation of an integer value - 4 bytes.

There are variations to the representation showed in Figure 3 due to the integer values can be represented in *Little Endian* or *Big Endian* format.

If the original variable has m observations, the size in bytes needed to represent all the observations (without compression) is: $Total\, bytes = Tb = m * 4$

If we consider that all 32 bits are not used to represent the values, there is a lot of wasted space.

In Figure 4, the gray area corresponds to space that is not used. Out of a total of $4 * 32 = 128\, bits$, only 16 are used, which represents 12.5% of the total storage used.

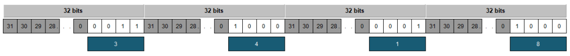

Figure 4: Example, representation of 4 values.

The central idea of this work consists in re-use the gray areas.

3.1 Minimum Number of Bits

Let $V = \{v_1, v_2, v_3, \ldots, v_{m-1}, v_m\}$ a categorical variable where the values $v_i \in D = \{x_1, x_2, x_3, \ldots, x_k\}$ (set of all possible values that variable V may take). It is required that the values of the set D are ordered from lowest to highest, this is $x_1 < x_2 < x_3 < \ldots < x_k$. From the previous definition, the values of a categorical variable are between x_1 and x_k.

The minimum number of bits needed to represent a value of the aforementioned variable corresponds to:

$$n = \left\lceil \frac{ln(x_k)}{ln(2)} \right\rceil \tag{1}$$

where $\lceil \cdot \rceil$ represents the smallest integer greater than or equal to its argument and is defined by $\lceil x \rceil = min\{p \in \mathbb{Z} | p \geqslant x\}$.

3.2 Maximum Number of Values Represented

Using Equation (1), we can determine the number of elements of the V variable that can be stored within an integer value (4 bytes):

$$N_V = \left\lfloor \frac{32}{n} \right\rfloor \tag{2}$$

where $\lfloor \cdot \rfloor$ represents the largest integer less than or equal to its argument and is defined by $\lfloor x \rfloor = max\{p \in \mathbb{Z} | p \leqslant x\}$.

With this solution it is possible that 32 bits were not used in total, leaving a number of these unused.

Table 1 shows a summary of the number of bits used to represent a certain number of categories. The first column shows the number of categories to represent (2, between 3 and 4, between 5 and 8, etc.), the second column shows the number of bits needed to represent the categories mentioned, finally the third column shows the total number of elements that can be represented within 32 bits.

This method generates a new subset V' that contains integer values, which in turn contain N_V elements of the original set. The number of elements in this set is given by:

$$Number\,of\,elements = m/N_v \tag{3}$$

Table 1: Amount of elements to represent.

Num. categories	Num. bits	Total elements
2	1	31
3 to 4	2	15
5 to 8	3	10
9 to 16	4	7
17 to 32	5	6
33 to 64	6	5
65 to 128	7	4
129 to 256	8	3
257 to 512	9	3

where m is the number of elements of the original set V and N_V is given by Equation (2).

3.3 Indexing Elements

To index an element within the original set V, it is necessary to double indexing the set V'. Let i be the index of the element to be searched within the original set V, the index i' on the set V' is given by:

$$p = \lfloor i/N_v \rfloor \; ; \; q = i \mod N_v \tag{4}$$

which represents the index on the 32-bit element to determine the actual value sought.

In summary, the element of the *i-th* position in the set V can be extracted from the set V' in the following way:

$$v_i = (V'_p \gg q * n) \;\&\; mask \tag{5}$$

where p and q are given by the Equation (4) and $mask = 111...1$ (sequence of n values 1, n is given by the Equation (1)). Figure 5 illustrates the above.

3.4 Algorithms

The algorithm represents n categorical values of a variable within 4-bytes. This means that to access the value of a particular observation, a double indexing is necessary: access the index of the integer value (4-bytes) that contains the searched element and next index within 32 bits to access the value.

The implementation of the proposed method uses a mixture of arithmetic operations and bitwise operations: Logical AND (&), Logical OR (|), Logical Shift Left ($\ll$), Logical Shift Right ($\gg$).

Algorithm 1 represents the algorithm to compress a traditional vector to the bit-to-bit format and algorithm 2 represents the algorithm to iterate over a compressed vector.

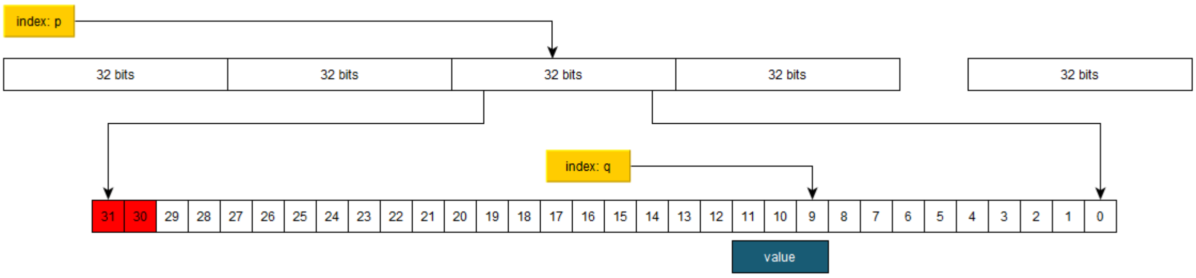

Figure 5: Indexing elements.

Algorithm 1: Compression algorithm.

Data: data, size, dataSize
Result: buffer, buferSize, elementsPerBlock

$elementsPerBlock \leftarrow 32/dataSize$;
$bufferSize \leftarrow ceil(size/elementsPerBlock)$;
$buffer \leftarrow new\ int[bufferSize]$;
$blockCounter \leftarrow 0$;
$elementCounter \leftarrow 0$;
for $i \leftarrow 0$ **to** $size - 1$ **do**
  $value \leftarrow data[i]$;
  if $blockCounter \geq elementsPerBlock$ **then**
    $blockCounter \leftarrow 0$;
    $elementCounter \leftarrow elementCounter + 1$;
  end
  $vv \leftarrow value \ll (dataSize * blockCounter)$;
  $buffer[elementCounter] \leftarrow buffer[elementCounter] \mid vv$;
  $blockCounter \leftarrow blockCounter + 1$;
end

Algorithm 2: Iteration algorithm.

Data: vector, size, dataSize

$elementsPerBlock \leftarrow 32/dataSize$;
$mask \leftarrow$ sequence of dataSize-bits with value $= 1$
for $index \leftarrow 0$ **to** $size - 1$ **do**
  $value \leftarrow vector[index]$;
  for $i \leftarrow 0$ **to** $elementsPerBlock - 1$ **do**
    $v \leftarrow (valor \gg i * dataSize)\ \&\ mask$;
    do something with v;
  end
end

4 EXPERIMENTS

This section presents the result of the compression of random generated vector and some known datasets. The memory consumption of the representation of the data in the main memory of a computer was analyzed, this value represents the amount of memory necessary to represent a vector of n-elements.

4.1 Random Vectors

In the first instance, several vectors of size 10^3, 10^4,..., 10^9 elements were generated. The test data set contains elements randomly generated in the range $[0, 120]$.

Table 2 shows the result of compressing different vectors using the proposed method.

Table 2: Memory consumption.

Size (# elements)	Memory consumption (Kb) Uncompressed (int)	Compressed
10^3	3.9	1
10^4	$3.9 * 10^1$	10^1
10^5	$3.9 * 10^2$	10^2
10^6	$3.9 * 10^3$	10^3
10^7	$3.9 * 10^4$	10^4
10^8	$3.9 * 10^5$	10^5
10^9	$3.9 * 10^6$	10^6

For the following tests we considered a vector of size $n = 10^9$ whose elements are integer values in the range of 0 to 120. The representation of each element corresponds to a 32-bit floating value, whereby the size in bytes to represent the vector is $total\ bytes = 10^9 * 4\ bytes$. This value represents 100% in the Figure 6 which shows the memory consumption for the vector representation mentioned above and the representation by 7 bits and 2 bits.

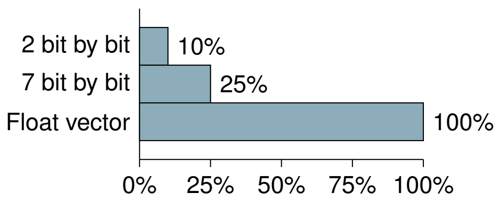

Figure 6: Vector size by memory consumption.

As you can see, the memory consumption for the *bit-by-bit* representation corresponds to 25% of the original value.

Compared with traditional compression methods (ZIP), Figure 7 shows the relationship between the original file, the compressed file with bit-to-bit and the compressed file with the linux ZIP utility.

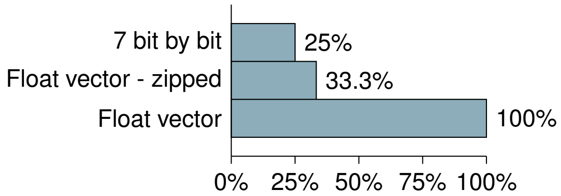

Figure 7: Compressed vector size.

Figure 8 shows the compression ratio between the generated file with the bit-to-bit algorithm and the same compressed file with the ZIP utility, as can be seen, the compression ratio is very high, which shows that the file generated with the proposed algorithm has a high level of compression.

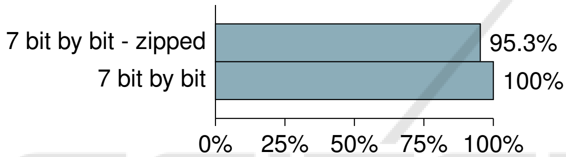

Figure 8: Bit to bit vector size.

4.2 Public Dataset

In the case of public datasets, the compression method was verified with the *US Census Data (1990) Data Set* [3] which contains 2458285 observations composed of 68 categorical attributes.

Table 3 shows the ranges of the 5 attributes with the highest values. As it is observed, the *iRemplpar* attribute has the biggest range $[0, 233]$, nevertheless it has 16 categories. The compression used corresponds to the number of categories of the attributes.

The *iYearsch* attribute has 18 categories. According to Section 3.2, the number of bits needed to represent the values of each column corresponds to $Number of bits = 5$ with which it is possible to represent 6 elements per block.

The number of bytes required to store the dataset without compression in memory [4] corresponds to $Tb = (2458285 * 68 * 4)\, bytes = 668653520\, bytes$. Thus, the amount of memory corresponds to $Tb = 637.68\, Mbytes$ [5].

[3]https://archive.ics.uci.edu/ml/datasets/US+Census+Data+(1990)

[4]Most ML libraries requires integer values

[5]R software shows 640 Mbytes of memory consumption

Table 3: US Census Data (1990) Data Set - Ranges.

N.	Variable	Min	Max	Max. value	Cats.
1	iRemplpar	0	223	223	16
2	iRPOB	10	52	42	14
3	**iYearsch**	0	17	17	18
4	iFertil	0	13	13	14
5	iRelat1	0	13	13	14
...	...	...	...	...	...

The number of bytes needed to store the dataset with compression in memory corresponds to: $Tbc = (2458285 * 68 * 4)/6\; bytes = 111442253,33\; bytes$ whereby the amount of memory corresponds to $Tbc = 106.28\, Mbytes$. Figure 9 shows the compression ratio of the test dataset.

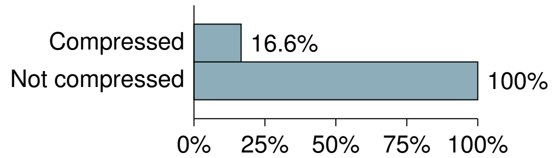

Figure 9: US Census Data (1990) Data Set - Memory consumption.

5 CONCLUSIONS

In this document we reviewed some of the traditional compression/encoding methods of categorical data. In general, we can conclude that the proposal made considerably reduce the amount of memory needed to represent the data prior to be processed (see Figure 7). The block decompression allows to process the dataset without the need to completely decompress it prior to the process, this allows to keep the dataset compressed in memory instead of its uncompressed version.

In the case of datasets with multiple columns, it was shown that selecting the column with the most categories provides a good compression ratio (see Section 4.2). This can be optimized by taking into account the appropriate size for each column as it would increase the compression ratio.

Future work may be proposed: (1) implement the Basic Linear Algebra Subprograms (BLAS) standard which defines low level routines to perform operations related to linear algebra which provide the basic infrastructure for the implementation of many machine learning algorithms and (2) implement compression with larger block sizes (eg 64 bits).

ACKNOWLEDGEMENTS

The authors would like to thank to Universidad Central del Ecuador and its initiatives *Proyectos Semilla* and *Programa de Doctorado en Informtica* for the support during the writing of this paper. This work has been supported with Universidad Central del Ecuador funds.

REFERENCES

Bruni, R. (2004). Discrete models for data imputation. *Discrete Applied Mathematics*, 144(1-2):59–69.

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

De Grande, P. (2016). El formato Redatam. *ESTUDIOS DEMOGRÁFICOS Y URBANOS*, 31:811–832.

Elgohary, A., Boehm, M., Haas, P. J., Reiss, F. R., and Reinwald, B. (2016). Compressed Linear Algebra for Large-Scale Machine Learning. *Vldb*, 9(12):960–971.

Elgohary, A., Boehm, M., Haas, P. J., Reiss, F. R., and Reinwald, B. (2017). Scaling Machine Learning via Compressed Linear Algebra. *ACM SIGMOD Record*, 46(1):42–49.

Gan, G., Wu, J., and Yang, Z. (2009). A genetic fuzzy k-Modes algorithm for clustering categorical data. *Expert Systems with Applications*, 36(2 PART 1):1615–1620.

Hashem, I. A. T., Anuar, N. B., Gani, A., Yaqoob, I., Xia, F., and Khan, S. U. (2016). MapReduce: Review and open challenges. *Scientometrics*, 109(1):1–34.

Huang, Z. and Ng, M. K. (1999). A fuzzy k-modes algorithm for clustering categorical data. *IEEE Transactions on Fuzzy Systems*, 7(4):446–452.

Ouyang, J., Luo, H., Wang, Z., Tian, J., Liu, C., and Sheng, K. (2010). FPGA implementation of GZIP compression and decompression for IDC services. *Proceedings - 2010 International Conference on Field-Programmable Technology, FPT'10*, pages 265–268.

Rai, P. and Singh, S. (2010). A Survey of Clustering Techniques. *International Journal of Computer Applications*, 7(12):1–5.

Roman-gonzalez, A. (2012). Clasificacion de Datos Basado en Compresion. *Revista ECIPeru*, 9(1):69–74.

Seshadri, V., Hsieh, K., Boroumand, A., Lee, D., Kozuch, M. A., Mutlu, O., Gibbons, P. B., and Mowry, T. C. (2015). Fast Bulk Bitwise and and or in DRAM. *IEEE Computer Architecture Letters*, 14(2):127–131.