Universiteit Leiden

FACULTY OF SCIENCE - Course 2019/ 2020

Advance Data Management for Data Analysis

1st Assignment
Encoding and Decoding

Authors:
Lisa Dombrovskij (s1504819)
Margherita Grespan (s2233150)
Juan de Santiago Rojo (s2302470)

---

1 Introduction

Throughout this report, five different encoding and decoding techniques will be described. These techniques have been developed and implemented in order to apply them as compression techniques over a group of data sets. Furthermore, this technique will encode and decode each data set provided as a stand-alone program in which the user can decide between encoding or decoding a data set selected by its name and file extension.

The outcome of encoding and decoding the data sets, provided with the developed techniques, will be presented and compared to each other. All code needed for its implementation has been developed using Python. Note that, as required, not only each of the technique's implementation will be explained, but also the code behind it. Furthermore, the necessary steps to run the program will also be elaborated upon.

2 Problem Statement

Compression has been used as a technique to increase performance in many database management systems (DBMS). As it reduces the size of data on disk and decreases seek times, it improves the performance. As a result, it increases the buffer pool rate and the data transfer rate. Compression can be seen as an optimization on disk storage.

When talking about data compression techniques, they can be divided into two broad groups: lossless compression, where the decoded data is identical to the encoded data, and lossy compression, which generally provide much higher compression than lossless compression, but allows the decoded data to be somehow different from the encoded data.

Lossless compression techniques involve no loss of information. When the data has been losslessly encoded, the original data can be recovered from the encoded data. Meanwhile lossy compression techniques may produce some loss of information. The lossy encoded data cannot be decoded exactly. As a trade off, these techniques provide much higher compression ratios than the lossless compression techniques and as a result, lossless compression is used for applications that do not tolerate differences between the original and decoded data, whereas lossy compression techniques are used in applications where the lack of exact recovery of the data can be tolerated, such as storing and transmitting speech.

For this assignment, we have been given a set of comma separated files. For encoding, this data must be read from disk and then written back to disk using five encoding formats. These encoding formats are: uncompressed binary, run length encoding, dictionary encoding, frame of reference encoding and differential encoding. They will be described in the upcoming section 4. Data type restrictions are applicable depending on what data type the file is, since some encoders do not apply for certain data types. This is specifically the case with binary encoding, frame of reference encoding and differential encoding, which will only be applied to integer types.

For decoding the data, after being encoded previously, it will be read from the encoded file and then shown as plain decoded text.

3 Data

The data sets provided are single-column comma-separated values (CSV) files, which use line-end as record separator. Note that each one of the data sets has one value per row. There are five different data types which will be encoded and decoded:

- "int8": 8-bit integer
- "int16": 16-bit integer
- "int32": 32-bit integer
- "int64": 64-bit integer
- "string": character string of arbitrary length

As an example of what the data looks like, the first few lines of an original file "l_comment-string.csv" is shown here:

```
egular courts above the
ly final dependencies & slyly bold
riously. regular, express dep
lites. fluffily even de
pending foxes. slyly re
arefully slyly ex
ven requests. deposits breach a
ongside of the furiously brave acco
unusual accounts. eve
nal foxes wake.
y. fluffily pending d
ages nag slyly pending
ges sleep after the caref
```

Original file example

Furthermore, the given data-sets have each different characteristics. For the integer data-type files, some have larger values, as for example *l_extendedprice-int64.csv* which value range goes from 90100 to 10494950, where as, *l_discount-int8.csv* only ranges from 0 to 10. The first file does obviously represent a significantly higher size and therefore requires higher processing time for compression.

Similarly, the string files contain different string structures. The textitl_returnflag-string.csv file only contains single letters, whereas the textitl_shipinstruct-string.csv contains multiple words per row, which sometimes are repeated. Not only that strings can be words but also numbers as for example, *l_shipdate-string.csv* which contains dates. That can also be repeated. The repetition of values in the files will affect the performance of the encoding techniques.

4 Encoding and Decoding

Throughout this section, the five compression techniques used for this assignment will be introduced and described. Some examples will be given in order to make more clear how each compression technique works. To give a complete picture of these kinds of compression we briefly discuss the role of bits in computer memory.

4.0.1 Bits

The memory used to store the encoded file, and the ability to decode the data are the most important aspects to consider when compressing a data-set. In computer memories data are stored in bits. The smallest amount of memory storage possible is a single bit. One bit can represent one of two values, where we use the convention that a single bit represents the value 0 or 1. Furthermore, bits and computer memory in general are represented by binary numbers.

To be more specific, in Python an integer vector can be defined by different data types. The allowed sizes are 8, 16, 32 and 64 which can be signed or unsigned[1].

In the "binary world", by convention, the sign of a number can be represented by the leftmost value, i.e. 0 for positive numbers and 1 for negative numbers. This means that if negative numbers have to be represented in binary numbers, the number of bits that can be used is diminished by 1. For example, if there are 8 bits available and a signed number has to be represented in those 8 bits, there will be seven bits which can represent the actual binary number, as the first bit will give the sign. The maximum binary number possible in 7 bits is 1111111, that in decimals is:

$$1111111 = 1 \cdot 2^6 + 1 \cdot 2^5 + 1 \cdot 2^4 + 1 \cdot 2^3 + 1 \cdot 2^2 + 1 \cdot 2^1 + 1 \cdot 2^0 = 127 \tag{1}$$

Hence, with 8 bits we can represent numbers from -128 to 127 (we are counting the 0 as a positive number) or from 0 to 255.

4.1 Uncompressed binary encoding and decoding

The above example given in equation 1 leads to the main method used for the uncompressed binary encoding technique. The technique is quite straight-forward, as it simply converts every decimal value in the data to a binary value.

In order to do so, the integer read from the file is divided by two. The integer quotient is the passed to the next iteration and the remainder is set as a binary digit, in ascending order, so that first iteration sets the far right binary digit in the encoded binary. These steps are repeated until the quotient is equal to zero. This is shown in this example, where the integer is 21 and the binary encoded result is 10101.

[1] Only natural numbers.

Division by 2	Quotient	Remainder	Bit #
21/2	10	1	0
10/2	5	0	1
5/2	2	1	2
2/2	1	0	3
1/2	0	1	4

For decoding the binary file, each binary number is read and convert into a decimal number. To do so, after reading the binary number and based on the position (n), each binary bit is multiplied by the power of 2^n. Where the right most number represents position $n = 0$. As it is shown in the following example, when 10101 is decoded following the described procedure, the outcome will be 21.

Binary #	1	0	1	0	1
Power of 2	2^4	2^3	2^2	2^1	2^0

$$\text{Decimal \#} = 1 * 2^4 + 0 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 = 21$$

4.2 Run length encoding and decoding

Run length encoding is an encoding technique that works well when the data contains a lot of repeating values. With this technique, repeating values are compressed into two values: the original value and the count. The count shows how many times the value was consecutively repeated in the original data. An example might be the following data:

aaaabbcccaaaaa.

With run length encoding, this data would be compressed as

a4b2c3a5.

For every pair of two elements in the encoded data, the first element gives the original character, and the second element gives the amount of times it was consecutively repeated in the data. In the above example, the character *b* shows why this technique works best with data that contains characters that repeat more than twice. The decoded part *bb* takes up two places in the original array and the encoded *b2* does so as well. If all characters do not show up consecutively, the encoded array will become twice as long. Only when a lot of repeating characters can be compressed into two characters each, is this technique beneficial.

The decoding, then, is very straight-forward. The encoded array is read in consecutive pairs of elements, repeating each original character n times, with n being the second element of each pair, i.e. the count.

4.3 Dictionary encoding and decoding

Dictionary compression is a widely used lossless data compression technique. In literature we can find different examples to how is possible to optimize this technique, e.g. Abadi, Madden, and Ferreira 2006.

Although, in this work we will implement a basic version of this compression algorithm.

As the name suggest, this technique stores all the unique elements in a data structure (*dictionary*) with the supplement of a *pointer vector*. The dictionary is created with as keys all the *unique* elements found in the database, and as values a unique integer between 0 and $n-1$ with n amount of unique elements in the database.

The pointer vector has the role to match the database with the data-set. Therefore, it has the same length as the data-set and is composed by the dictionary's keys in the order the elements represented by those are found. We can see now how this vector works as an encoder, namely it creates a connection between the dictionary (encoded data) and the database (decoded data).

To make clear how the compression works an example will be shown. Consider a database consisting of integers, e.g. $1, 4, 7, 1$. Our dictionary will have three entries, the unique keys, namely $1, 4, 7$ with values respectively,$0, 1, 2$. The values represent the encoded version of each unique element in the original data-set.

The encoder vector will then be $0, 1, 2, 0$. It is evident that for a data-set consisting of small numbers, this technique will not make a significant difference on the amount of storage the file will take up. However, when a data-set contains, for example, large number values or long strings, the integer values that are used for the encoding will make a difference.

When decoding data that has been encoded using this method, the dictionary is used to relate the encoded values back to their original values. In practice, this means the dictionary is flipped; The original values, the keys, become the values and the encoded values become the keys. That way, retrieving the original element from an encoded element is as simple as calling the dictionary using that element (*dict[element]*) and it will return its original value.

4.4 Frame of reference encoding and decoding

In physics, a *frame of reference* (f) is defined as an abstract coordinate system on which all the calculations are based. More specifically, a frame of reference is used to standardize measurements within a frame.

The frame of reference encoding technique is used for numerical data-sets with some kind of correlation in it, i.e. the successive elements in the data-sets are not too different from each other. Goldstein, Ramakrishnan, and Shaft 1998 propose a new compression algorithm that can be easily added to the file management layer of a DBMS. This approach is based on two levels, a *page level compression* and a a *file level compression*. In this work we focus on the page level one which permits to take advantage of common information in between a page. The common information is exactly the frame of reference. Using this, data can be compressed in reference to a unique value.

As said before, this technique encodes values based on a frame of reference, namely an element of the data-set. This can be the first element of the data-set, the mean or median of the data-set, or any other value decided upon. The encoded value is the numerical difference between f and every element of the data-set.

At the start of our encoding, the frame of reference is set to be the mean value of the data-set and the subsequent numbers (n_i with i index which goes from 0 to length of the data-set) are stored in the encoded file as their difference with the frame, (n_i-f). Ideally, the number of bits required to represent

n-f is smaller, thus the compressed data will require less storage space. Choosing the amount of bits to represent an encoded data-set is not straightforward. When the trend of the data in the database is known it easier to decide a suitable amount of bits. In order to know the range in the data-set one has to find the maximum and the minimum in it, this requires time and CPU usage. So, one can also opt for a fixed number of bits to represent the encoded data-set, i.e. an 8-bit Python array. Furthermore, we do not know if a database shows a monotonic rate[2], so we assume that it is possible that the difference between the frame of reference and an element in the data will be negative, meaning that the number of bits used to represent each number is diminished by one, see section 4.0.1 for further explanations.

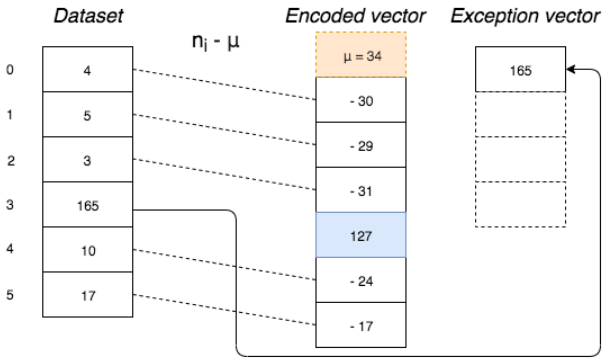

Figure 1: Frame Of Reference Encoding.

As already mentioned, we are not checking the range the data-set covers so with a fixed number of bits we expect some exceptions. More specifically, a value n_i-f might be bigger than the the admissible values that can be stored in 8 signed bits. In this case, an escape code is used to signify to the decoder that an exception was encountered. This escape code will by an 8-bit value with each bit set to 1. This means that the number 127, which is represented by eight 1's in a signed 8-bit value, is also an exception. The escape code is added to the encoded vector as an element, instead of the exception, and the exception is added to the exception vector. This way, the decoder can correspond the first encountered escape code to the first exception in the exception vector, the second encountered escape code to the second element in the exception vector, and so on.

To clarify in Figure 1 we show an example. The frame of reference is set to be the mean of the data-set, in this case this is 34. The first encoded element is the difference between the first element in the data-set and the frame of reference. In this case, $4 - 34$ makes -30. This method is continued until an exception is encountered, in this case element with index 3 in the data-set. The difference between this element (165) and the frame of reference (34) is too big to store in 8 bits. Thus, the exception code is added to the encoded vector in place of this difference, and the exception is added to the exception vector.

When decoding an array that is encoded using frame of reference encoding, the decoder reads the first element of the exception vector to be the frame of reference. It then starts reading the encoded 8-bit vector, summing the frame of reference to each element to obtain the decoded value, until the first escape code (the number 127) is encountered. Instead of adding the frame of reference to the escape code, the decoder adds the first element of the exception vector to the decoded vector. When encountering the next escape code, it inputs the second element of the exception vector, and so on.

[2]the numbers are only increasing or only decreasing

4.5 Differential encoding and decoding

Differential encoding is used in numerical samples which show correlation in between the elements. They are encoded using the difference between the element and the previous element.

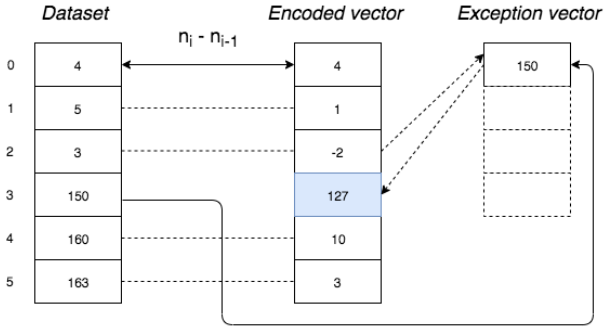

Figure 2: Differential Encoding.

This technique is similar to the frame of reference encoding described in section 4.4. The reasoning is similar: in order to get the data compressed, the difference between successive numbers is stored in an encoded array. This way, a smaller amount of memory, i.e. a smaller amount of bits, is used to save the data.

The difference between this approach and the reference encoding is that the difference is not computed in relation to a frame of reference, but to the precedent number in the data-set. In Figure 2 an example is shown. The first value in the exception vector is just the first value in the data-set. The first value in the encoded vector is the difference of the second element in the dataset with the first element, the third is the difference of the third with the second element, and so on. This process can be expressed with a simple equation:

$$n_{enc,i} = n_{data-set,i} - n_{data-set,i-1} \tag{2}$$

However, just like with frame of reference encoding, exceptions are possible. If the difference between two subsequent values is not representable in 8 bits an escape code is used to tell the decoder that the value it is going to encounter is not encoded in respect to the one before as it normally would be and can be found in the exception vector. The escape code, as with the frame of reference encoding, is the number 127, i.e. all bits set to 1. After the exception the process continues in the same manner explained previously.

The data compression with differential encoding is based on the idea that if data in a data-set are following a trend, the difference between two numbers requires less space in memory than the numbers themselves. Thus, this would work well on time series, for example.

For this approach the decoding is easy to implement. As we can see in fig. 2 the first element of the decoded vector is not compressed. We have a starting point, n_0. To go back to the original data-set we simply sum to subsequent numbers. Using a formal equation we can say that the element n_i - with i bigger than 0 - can be decoded as:

$$n_{dec,i} = n_{enc,i-1} + n_{enc,i} \tag{3}$$

When the next exception is encountered, the decoder simply takes the next element in the exception vector as the decoded value.

5 Implementation

In this section a brief overview of the code used to encode and decode the files will be given. The objective of the code is to ask the user to input the filename, datatype and whether the user wants the file to be encoded or decoded. If the user wants the file to be encoded, the program should read the file, encode it using all applicable encoding techniques, and store the encoded files to the disk using the correct extensions. If the user wants the file to be decoded, the program should read the correct encoded file and decode it, outputting the plain decoded text to the terminal screen.

5.1 Code Structure

The code is made up of three classes: *read_input*, *encode* and *decode*. The *read_input* class is dedicated to reading the input of the user. It first asks the user to input either *en* for encoding or *de* for decoding, then for the filename and lastly the datatype. It checks the three inputs. If the input to the first question is not *en* or *de*, it is asked again. When a non-existing filename is given by the user, the program asks the user for the filename again and if the datatype given by the user does not match the datatype that can be found in the filename, the user has to input the datatype again. This way, the program makes sure all of the input is correct before starting the encoding or decoding process. Once the class has the user input, it reads the input file.

The *encode* class takes care of all of the encoding. It takes the input file and, based on the datatype, encodes it using the applicable techniques. All datatypes are run length encoded and dictionary encoded, only for integer type data is binary encoding, frame of reference encoding and differential encoding is performed. Once all of the encoding is done, the program stores the encoded files to the disk using the following extensions: `.bin, .for, .dif, .rle` and `.dic` for binary encoding, frame of reference encoding, differential encoding, run length encoding and dictionary encoding respectively.

The *decode* does the decoding. It looks at the extension of the given filename to see what type of encoding was used and then decodes the file using the corresponding decoding technique. Once the file is decoded, it outputs the decoded data to the screen in plain text.

5.2 Running the Program

In order for a user to run the program, they only need the files and the Python script named *script.py*. This script should be in a directory along with a folder named *ADM-2019-Assignment-1-data-T-SF-1* in which all csv files are stored. The names of the csv files have to have the following format: *l_[name]-[datatype].csv*. With possible data-types being int8, int32, int64 or string. Then, from a terminal window, the script can be run by typing *python script.py*, assuming the user has Python installed.

6 Results

Throughout this section, the result after encoding each of the data-sets with the corresponding compression technique will be displayed.

To begin with, the original dataset sizes are shown in Table 1. Here, the size in bytes and in megabytes (MB) is displayed for each of the given datasets. Note that this sizes are then used for calculating the compression rate of each technique over each original file. It is important to evidence that in a csv file is possible to store only strings, consequently the data types are 'lost' when saved. In tab.1 the sizes shown are the estimates of the actual sizes of the data in relation to the datatype. In this way, for example, if we consider two files containing the same elements, where for sake of simplicity those are integers, the file saved with an int8 datatype will require less memory than an int32.

File name	Size (bytes)	Size (MB)	File name	Size (bytes)	Size (MB)
l_comment-string.csv	164998424	151,63	l_quantity-int64.csv	16924405	366,29
l_commitdate-string.csv	66013365	57,23	l_quantity-int8.csv	16924405	45,79
l_discount-int64.csv	12548245	366,29	l_receiptdate-string.csv	66013365	57,23
l_discount-int8.csv	12548245	45,79	l_returnflag-string.csv	12002430	5,72
l_extendedprice-int32.csv	47237224	183,14	l_shipdate-string.csv	66013365	57,23
l_extendedprice-int64.csv	47237224	366,29	l_shipinstruct-string.csv	78007624	68,67
l_linenuer-int32.csv	12002430	183,14	l_shipmode-string.csv	31718249	24,53
l_linenuer-int8.csv	12002430	45,79	l_suppkey-int16.csv	29341797	91,57
l_linestatus-string.csv	12002430	5,72	l_suppkey-int32.csv	29341797	183,14
l_orderkey-int32.csv	46898191	183,14	l_tax-int64.csv	12002430	366,29
l_partkey-int32.csv	38675408	183,14	l_tax-int8.csv	12002430	45,79

Table 1: Original dataset sizes

As explained in section 4, some compression algorithms only work for specific datatypes (see data in section 3). As an overview, the file extensions of the encoded files are shown again below, along with the corresponding encoding technique.

- `.bin`: Uncompressed Binary Format, described in section 4.1 which accepts only integer type data.
- `.rle`: Run-length encoding, section 4.2.
- `.dic`: Dictionary encoding, section 4.3
- `.for`: Frame of reference encoding, section 4.4, which works with integer types only.
- `.dif`: Differential encoding, section 4.5, which also works with integer types only.

As requested, the compression rate has been computed as the rate between the compressed file size and the original file size. If the rate is higher than 1, it means that the compression was not efficient, as the compressed file is larger than the original. Meanwhile, if the compression rate is lower than 1 this will mean that the compression of the file done by a certain technique was efficient, providing a smaller file size.

This compression rate can be found in column shown in Table 2. Here each one of the encoded data-sets, after being compressed by the corresponding encoding technique, is shown with its size in bytes and MB. Note that due to space constraints, this Table 2 has been placed in the Appendix.

7 Conclusion & Discussion

Throughout this section, the conclusions derived from this assignment will be presented. Note from Table 2, placed in the Appendix, that a significant number of encoded files have lower compression rates than 1. As mentioned in Section 6, if the compression rate is lower than 1, the encoded file is lower in size than the original file. Specifically, 77% of the encoded files show this, whereas the 23% left, have higher compression rates than 1, meaning that the encoding techniques do not perform as expected. It is important to note how the *run-length encoding* technique is responsible for most of the unsuccessful compression, especially when encoding string data-type files. Looking at those specific files, it becomes evident that there are very few repeating elements in the datasets. In general, if a file with no repeating elements is encoded using this specific run-length encoding technique, the number of elements in the file will simply double. Clearly, this technique will only work on data that has enough repeating values. More precisely, twenty-two files have been encoded with the run length encoding, eight have a compression rate higher than 1 and one has a rate equal to 1.

On the other hand, the *frame of reference* and *differential encoding* are the most successful ones. Each technique created 12 compressed files and all of them have a compression rate smaller than 1. We also emphasize the similarity between these approaches. As seen in section 4.4 and 4.5 the idea behind the techniques is comparable. The one file out of 12 with a different rate between the two techniques is *l_orderkey-int32.csv*. It is easy to figure out why the differential encoding worked better. The file in question is composed of a list of increasing integers. Moreover, the dif encoding requires less memory since every number is encoded in relation to the previous one. Instead, the for encoded the file with respect to the mean.

We observe that, as expected the *uncompressed binary encoding* is never bigger than 1.

Another notable occurrence is that the *dictionary encoding* of the file *l_comment-string.csv* was unsuccessful. This is signified in Table 2 by having all 0 values. The file contains 6001215 elements out of which 4580667 are unique. As the dictionary encoding creates a dictionary with a key for every unique element in the dataset, this results in a very large dictionary and subsequently an extremely large encoded file. This shows how this encoding technique works best on data with a relatively small number of unique values. Furthermore, sometimes the dictionary encoding results in a compression rate of 1.0, because the data already contains only small values. Specifically, in the twenty-one files encoded with the dictionary encoding, four have a compression rate equal to 1 and three higher than 1.

Appendix A Results Table

File name	Size (bytes)	Size (MB)	Rate	File name	Size (bytes)	Size (MB)	Rate
l_comment-string.dic	0	0	0	l_partkey-int32.dif	29975953	28,59	0,16
l_comment-string.rle	2064415710	1920	12,7	l_partkey-int32.for	29975726	28,59	0,16
l_commitdate-string.dic	48364313	46,12	0,81	l_partkey-int32.rle	48009715	45,79	0,25
l_commitdate-string.rle	474160238	452,19	7,90	l_quantity-int64.bin	48009881	45,79	0,13
l_discount-int64.bin	48009881	45,79	0,13	l_quantity-int64.dic	48012305	45,79	0,13
l_discount-int64.dic	48010403	45,79	0,13	l_quantity-int64.dif	6001467	5,72	0,02
l_discount-int64.dif	6001467	5,72	0,02	l_quantity-int64.for	6001468	5,72	0,02
l_discount-int64.for	6001468	5,72	0,02	l_quantity-int64.rle	94096115	89,74	0,24
l_discount-int64.rle	87286131	83,24	0,23	l_quantity-int8.bin	48009881	45,79	1,00
l_discount-int8.bin	48009881	45,79	1,00	l_quantity-int8.dic	48012019	45,79	1,00
l_discount-int8.dic	48010375	45,79	1,00	l_quantity-int8.dif	6001467	5,72	0,12
l_discount-int8.dif	6001467	5,72	0,12	l_quantity-int8.for	6001468	5,72	0,12
l_discount-int8.for	6001468	5,72	0,12	l_quantity-int8.rle	11762157	11,22	0,25
l_discount-int8.rle	10910909	10,41	0,23	l_receiptdate-string.dic	48376985	46,14	0,81
l_extendedprice-int32.bin	68179768	65,02	0,36	l_receiptdate-string.rle	477318878	455,21	7,95
l_extendedprice-int32.dic	102177326	97,44	0,53	l_returnflag-string.dic	48010150	45,79	8,01
l_extendedprice-int32.dif	30005405	28,62	0,16	l_returnflag-string.rle	33593405	32,04	5,60
l_extendedprice-int32.for	30005482	28,62	0,16	l_shipdate-string.dic	48372953	46,13	0,81
l_extendedprice-int32.rle	48009867	45,79	0,25	l_shipdate-string.rle	477092238	454,99	7,95
l_extendedprice-int64.bin	68179768	65,02	0,18	l_shipinstruct-string.dic	48010406	45,79	0,67
l_extendedprice-int64.dic	105912871	101,01	0,28	l_shipinstruct-string.rle	611827846	583,48	8,50
l_extendedprice-int64.dif	30005405	28,62	0,08	l_shipmode-string.dic	48010550	45,79	1,87
l_extendedprice-int64.for	30005482	28,62	0,08	l_shipmode-string.rle	288096973	274,75	11,20
l_extendedprice-int64.rle	96019571	91,57	0,25	l_suppkey-int16.bin	48009881	45,79	0,50
l_linenumber-int32.bin	48009881	45,79	0,25	l_suppkey-int16.dic	48569280	46,32	0,51
l_linenumber-int32.dic	48010242	45,79	0,25	l_suppkey-int16.dif	29400121	28,04	0,31
l_linenumber-int32.dif	6001467	5,72	0,03	l_suppkey-int16.for	29391022	28,03	0,31
l_linenumber-int32.for	6001468	5,72	0,03	l_suppkey-int16.rle	96009491	91,56	1,00
l_linenumber-int32.rle	46296507	44,15	0,24	l_suppkey-int32.bin	48009881	45,79	0,25
l_linenumber-int8.bin	48009881	45,79	1,00	l_suppkey-int32.dic	48589280	46,34	0,25
l_linenumber-int8.dic	48010227	45,79	1,00	l_suppkey-int32.dif	29400121	28,04	0,15
l_linenumber-int8.dif	6001467	5,72	0,12	l_suppkey-int32.for	29391022	28,03	0,15
l_linenumber-int8.for	6001468	5,72	0,12	l_suppkey-int32.rle	48004827	45,78	0,25
l_linenumber-int8.rle	11574249	11,04	0,24	l_tax-int64.bin	48009881	45,79	0,13
l_linestatus-string.dic	48010073	45,79	8,01	l_tax-int64.dic	48010315	45,79	0,13
l_linestatus-string.rle	13210797	12,6	2,20	l_tax-int64.dif	6001467	5,72	0,02
l_orderkey-int32.bin	67136420	64,03	0,35	l_tax-int64.for	6001468	5,72	0,02
l_orderkey-int32.dic	135012260	128,76	0,70	l_tax-int64.rle	85362963	81,41	0,22
l_orderkey-int32.dif	6001467	5,72	0,03	l_tax-int8.bin	48009881	45,79	1,00
l_orderkey-int32.for	30005322	28,62	0,16	l_tax-int8.dic	48010301	45,79	1,00
l_orderkey-int32.rle	12000163	11,44	0,06	l_tax-int8.dif	6001467	5,72	0,12
l_partkey-int32.bin	48009881	45,79	0,25	l_tax-int8.for	6001468	5,72	0,12
l_partkey-int32.dic	59609660	56,85	0,31	l_tax-int8.rle	10670513	10,18	0,22

Table 2: Encoded data-sets Sizes and Compression Rates

References

Abadi, Daniel, Samuel Madden, and Miguel Ferreira (2006). "Integrating Compression and Execution in Column-oriented Database Systems". In: *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*. SIGMOD '06. Chicago, IL, USA: ACM, pp. 671–682. ISBN: 1-59593-434-0. DOI: 10.1145/1142473.1142548. URL: http://doi.acm.org/10.1145/1142473.1142548.

Goldstein, Jonathan, Raghu Ramakrishnan, and Uri Shaft (1998). "Compressing Relations and Indexes". In: *Proceedings of the Fourteenth International Conference on Data Engineering*. ICDE '98. Washington, DC, USA: IEEE Computer Society, pp. 370–379. ISBN: 0-8186-8289-2. URL: http://dl.acm.org/citation.cfm?id=645483.656226.