Compression of Very Sparse Column Oriented Data

Vinícius Fulber Garcia[1], **Sérgio Luís Sardi Mergen**[1]

[1]Universidade Federal de Santa Maria

vfulber@inf.ufsm.br, mergen@inf.ufsm.br

Abstract. *Column oriented databases store columns contiguously on disk. The adjacency of values from the same domain leads to a reduced information entropy. Consequently, compression algorithms are able to achieve better results. Columns whose values have a high cardinality are usually compressed using variations of the LZ method. In this paper, we consider the usage of simpler methods based on run-length and symbols probability in scenarios where datasets are very sparse. Our experiments show that simple methods provide promising results for the compression of very sparse datasets where the delimiter is the predominant symbol.*

1. Introduction

Column oriented databases are not new. However the recent interest in NoSQL databases has lead to an increasing amount of research in this area. Differently from traditional relational databases that store rows contiguously on disk, the column oriented approach stores columns contiguously on disk. This physical organization allows faster access for queries that require a small amount of columns, mainly because less pages need to be load from disk into memory (Matei e Bank, 2010). Another advantage of the columnar approach over the row approach is data compression. In a column oriented database all information stored in a page belong to the same domain and data type. The information homogeneity leads to a reduced information entropy, and this fact can be explored by data compression algorithms (Abadi et al., 2006).

Compression in column oriented databases is achieved in several different ways. If cardinality is low (a small number of unique values exists), lightweight methods are normally used. Examples include dictionary encoding, RLE and bitmap indexing. If cardinality is high, variations of the heavyweight LZ compression method are normally used. Heavyweight methods are more flexible than lightweight ones. They are generally suited when data contains a certain degree of redundancy, since they are able to encode redundant parts effectively, achieving good compression ratio associated with an acceptable response time.

There is a special case of high cardinality columns where the underlying datasets are very sparse. Besides the patterns that are naturally found when the column values are stored in adjacent positions, the sparsity may also contain additional patterns composed of sequences of adjacent undefined values. Multiple occurrences of these sequences lead to redundancy, which opens an opportunity for compression. Methods based on LZ are naturally potential candidates to explore sequences of undefined values. However, it is an interesting perspective to investigate how different (and simpler) compression strategies handle such patterns.

In this paper we focus on the compression of very sparse datasets. Our goal is to verify how an LZ compression method behaves when compressing a significant number of long runs of the delimiter. We also intend to verify whether using simpler methods compensate in those cases, both in terms of compression ratio and compression/decompression speed.

This paper is organized as follows: Section 2 presents a motivational example. Sections 3 reviews compression methods relevant to our work: Section 3.1 and 3.2 focus on the methods called Run Length Encoding (RLE) and Entropy Encoding (EC), and Section 3.3 reviews concepts related to the LZ family of compression methods. For each of the aforementioned methods we outline in general terms how the sparsity is explored to achieve compression. The final part reports the experimental results and presents our concluding remarks.

2. Motivational Example

Figure 1 compares two distinct physical organizations of a fictitious PROJECT table composed by columns YEAR, STATUS and COMMENT. The NSM (N-ary Storage Model) page layout is employed by conventional relational databases that store records contiguously on disk. On the other hand, the DSM (Distributed Storage Model) page layout is employed by databases that store columns contiguously on disk. As we can see, a NSM page keeps whole records, whereas a DSM page keeps values from single columns (Ailamaki et al., 2001). The DSM design subsumes strategies used by modern columnar databases, such as MonetDB and Vertica (Lamb et al., 2012).

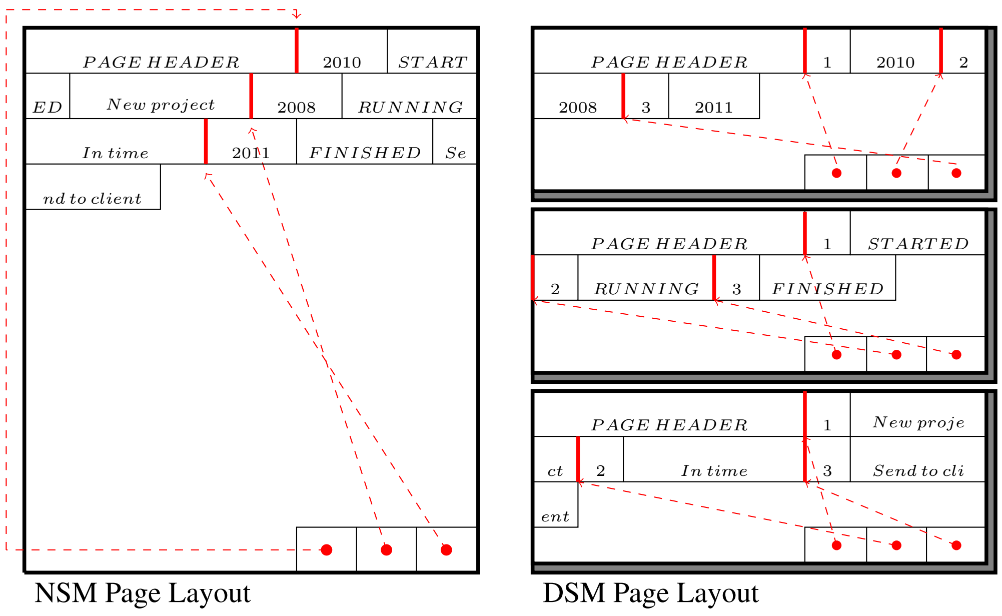


Figure 1. Three different page layouts

In both page layouts, the data grow from one side and auxiliary vectors grow from the other side. The vectors contain offset information, indicating where a record begins (in the case of NSM) or a field begins (in the case of DSM). Presence vectors could also be informed to indicate whether the value of a field is unknown. To simplify, the columns of our fictitious table do not accept null values. The blank spaces illustrated in the example correspond to empty fields, which are different than unknown.

Several lightweight compression methods are useful when values of the same column are stored contiguously in a page (Zukowski et al., 2006). For instance, fields from the YEAR column could be compressed using frame of reference, a method that codes values as a difference to a reference value (the average year). On the other hand, low cardinality columns such as STATUS could be compressed using dictionary encoding, a method that codes values as indexes that point to a dictionary where all possible values are mapped. Other useful techniques when cardinality is low include packing values into bytes to allow byte aligned dictionary indexing, run length encoding for sorted columns and bitmap encoding that uses different bitmaps to each possible distinct value.

The beauty of some lightweight methods (like Dictionary Encoding and Frame of Reference) is that they allow direct access to random fields, without the need to decompress the previous fields. In some cases it is also possible to operate directly on compressed data. For instance, it is easy to compare if a given compressed field is greater than a specific year by using the reference value as part of the comparison.

However, not all columns support this kind of compression. Take for instance the COMMENT column. Comments are usually variable length sentences written in natural language. The lack of regularity in textual data types inhibits the usage of simple encoding schemes. The same limitation would apply to the YEAR and STATUS columns if the data domain was more relaxed. For instance, if text is allowed as part of the content of YEAR, the presence of any non-numeric character would break the frame of reference method. Similarly, if the possible values for STATUS are not restricted, data can become too skewed for a dictionary encoding to succeed.

The presence of irregular values demands the usage of heavyweight compression methods, where the purpose is to apply a more sophisticated approach (and usually more time consuming) to reduce the number of bits needed to encode a message. Heavyweight methods do not allow direct access to individual fields from a compressed message. However, they may be the only solution if one wants to reduce the space occupied by the values from a column.

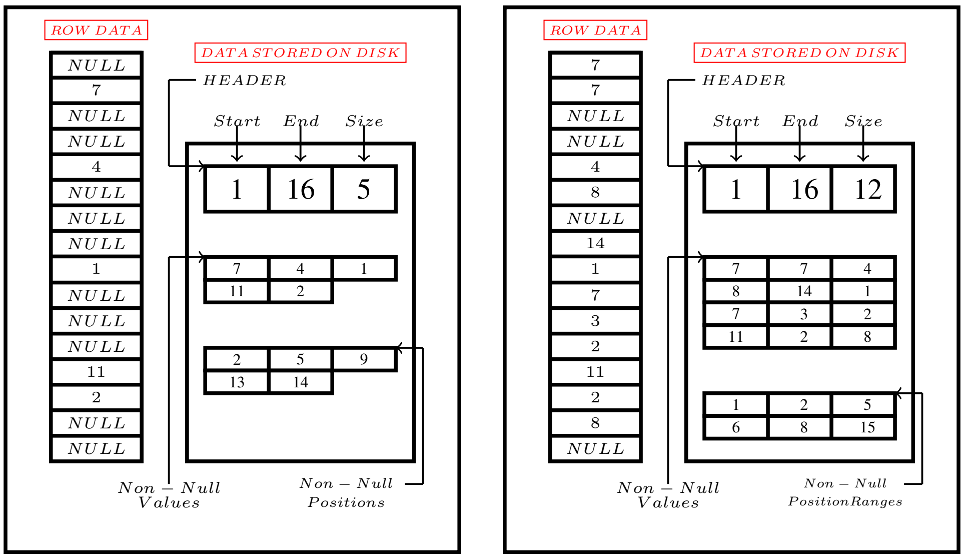
It is important to observe that the usage of compression methods that lack the ability to work directly on compressed data implies that offset vectors are useless. Instead, we can tell one field from the other by a reserved symbol used as a delimiter. For instance, assuming the semi-colon (;) is the delimiter, the status fields presented in Figure 1 could be serialized in a DSM based page as 'started;running;finished'.

The presence of delimiters in very sparse datasets form redundancy patterns that can be explored to increase compression. For instance, assume as our running example a new collection of status fields whose possible values are 'a' and 'b'. Also, assume all values are undefined, except two, as indicated below:

; ; ; a ; ; ; b

Observe that adjacent undefined status form runs of the delimiter symbol. This kind of pattern is naturally compressed by heavyweight methods. What we argue in this paper is that some lightweight methods could be useful for very sparse datasets as well.

Before going any further, we observe the existence of null suppression techniques useful for encoding very sparse datasets composed by a significant amount of null fields. For instance, Abadi et al. (2007) propose innovative page layouts to be used depending on the sparsity of data, as Figure 2 shows. If data is very sparse, null values are not represented at all. Instead, the pages keep a list of the positions where data is not null and another list with the values contained in those positions. This page layout was proposed as part of the C-Store database (the academic version of the Vertica database). The pitfall of the approach is that it was designed to work with fixed length columns in order to enable trivial vectorization. On the other hand, in this paper we consider variable length columns that could not be smoothly accommodated in a vector.



Sparse Dataset Dense Dataset

Figure 2. A page layout for storing very sparse fields

To restate, in this paper we care about very sparse datasets from variable length skewed/irregular columns that are not properly handled by null suppression techniques. Next section shows how welterweight and heavyweight methods handle the redundancy patterns formed when the delimiter appears in consecutive positions.

3. Compression Methods

This Section presents state of the art lossless compression methods for texts. The methods first appeared decades ago, and their essence remained the same throughout the years. In what follows we present the original techniques, discussing how sparse datasets can benefit from the core ideas.

3.1. Run Length Encoding

Run Length encoding (RLE) is one of the most simple compression methods. Basically, the purpose is to replace runs with codes. Many variations of this technique exist. One of them is to encode every run, regardless of its length. In this case the run is replaced by a code composed by the symbol and the count of how many times it repeats.

Another approach is to only encode runs whose length is higher than a predefined threshold. An escape symbol is needed to indicate that the coding scheme is about to be used. If no symbol can be reserved to be used as escape, one trick is to enter the coding scheme after the symbol repeats itself a specific number of times.

The nature of data in very sparse column oriented datasets allows a more proper encoding scheme. In most cases no symbols will be part of a run except for the delimiter. Therefore, one strategy is to encode the delimiter as the symbol itself followed by the count of how many other delimiters belong to the run. All other symbols are copied without encoding.

Figure 3 shows how the running example message would be encoded using this scheme. The message with nine bytes is compressed within six bytes, which yields a compression ratio of 33%, or 5,3 bits per code (bpc). Observe that the symbols are encoded as characters and the counting is encoded as decimal numbers. Also, the codes have a fixed 8 bit representation to allow byte aligned processing.

message	;	; ;	a	;	; ; ;	b
RLE	;	2	a	;	3	b
bytes	00111011	00000010	01100010	00111011	00000011	01100011

Figure 3. Encoding the running example message with RLE

Despite the simplicity, this technique is useful in many different compression scenarios, as part of more sophisticated methods, such as JPEG, to perform lossy image compression (Skodras et al., 2001) and BWT, to perform lossless text compression (Burrows e Wheeler, 1994). Using RLE in isolation is worthy only in very specific scenarios, where very long runs prevail. The experiments in Section 4 discusses how sparse the column oriented files should be so RLE outperforms other methods.

3.2. Entropy Encoding

In information theory, the entropy of a message is the minimal amount of bits necessary to represent the symbols of the message. This number of bits is strongly related to how predictable the message is. A low entropy means it is more easy to predict what symbols are more likely to appear, so less bits are needed to encode a message. For instance, files where one symbol is much more common than the others (like the delimiter in very sparse columns oriented files) have a low entropy, since we can predict that symbol will appear in most positions inside that file.

Two of the most popular compression methods based on entropy are the Huffman coding (Huffman et al., 1952) and arithmetic coding (Witten et al., 1987). The purpose of entropy encoding (or statistical encoding) methods is to reduce the number of bits used to encode symbols that occur frequently. The way these methods achieve this is quite different, as we explain next.

Huffman Coding: The Huffman coding assigns to each symbol of the message an unique and unambiguous sequence of bits, reserving the smaller sequences to the most frequent symbols. A binary tree is used to create that mapping. The construction of the tree is fairly simple. First all symbols are transformed into nodes and each node is assigned with its corresponding frequency. Than, a new node is created to contain as children the two less frequent nodes. The frequency of the newly created node is the sum of the child's frequencies. The process repeats until all nodes (but one) have a parent. The node without a parent becomes the root. Bits are assigned to all edges, using zero and one to left and right edges respectively.

Figure 4 shows the tree generated based on the running example message. Observe that coding the delimiter requires a single bit. After compression, the nine characters of the file are transformed into a 11 bit sequence (00010000011), giving a compression ratio of approximately 85%, or 1,22 bpc.

Symbol Mapping

Symbol	Frequency	bit sequence
;	7	0
a	1	10
b	1	11

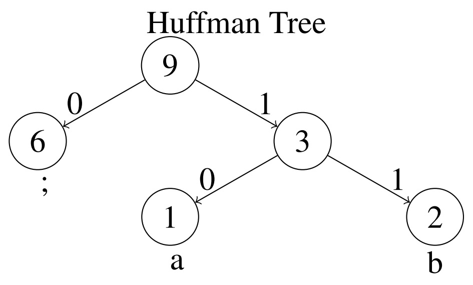


Figure 4. The Huffman tree construction based on the running example message

Arithmetic Coding: Unlike huffman code, in the arithmetic coding method there is no unique bit sequence that determines each symbol. Instead, the bits lead to a value that indicates the probability of occurrence of that exact sequence of symbols being compressed.

The probability ranges from zero to one. For sufficiently long messages, the probability would require a floating point precision higher than computers are able to express. To circumvent this technical problem, a fixed point precision value is used. When the value is about to overflow, the most meaningful bits are flushed out and the value is shifted to the right.

Figure 5 illustrates how the encoded probability is updated as the symbols are processed. Initially the probability of each symbol is divided into a scale ranging from zero to one and the probability range is updated as the symbols are processed. To simplify, the first case shows the probability distribution after the nine symbols of the message were processed. At this point, the probability of finding another delimiter is 77%. If the delimiter is indeed found, the probability is updated as demonstrated in the second case. As it shows, the probability of finding another delimiter after the first nine symbols drops to 60%. The probability keeps being updated as the remaining symbols are processed, and eventually the most significant bits are flushed.

In comparison to Huffman code, the arithmetic coding is better at approximating the entropy of the message. Moreover, it simplifies the process of adaptation, where the probability of each symbol is updated as the symbols are being read. Huffman encoding, on the other hand, needs to apply a preprocessing step over the whole sequence in order to determine the individual frequencies, or use complex tree manipulation operations in order to adjust the tree topology.

To conclude, the example shows that the entropy encoding resulted in a much better compression ratio if compared to RLE. The reason is mainly related to the size of the message, too small for RLE to be effective. Our experiments in Section 4 present a more proper analysis that verifies how the methods behave in terms of compression ratio and compression/decompression speed when working with very sparse (and long) datasets.

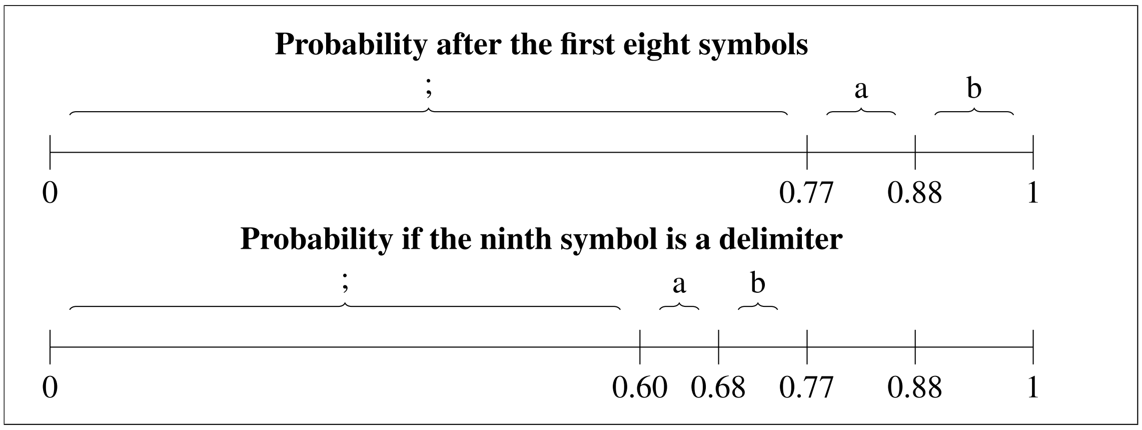

Probability after the first eight symbols
;
a
b
0
0.77
0.88
1
Probability if the ninth symbol is a delimiter
;
a
b
0
0.60
0.68
0.77
0.88
1

Figure 5. Encoding probabilities based on the running example message

3.3. Lempel-Ziv

Compression methods part of the LZ family are based on an idea proposed by Ziv e Lempel (1978). The method is a kind of dictionary encoding, where the dictionary is a sliding window composed by the most recently processed symbols of the message. The method checks a lookahead buffer if there is a sequence of symbols starting at the current symbol that can be dictionary encoded.

Many works propose variations related to the way the sequences found in the dictionary are encoded. In what follows we discuss the idea whose specification was later standardized: A sequence is encoded as a pair formed by an index value and an length value. The index refers to an offset inside the sliding window where the sequence being encoded can be found. The length refers to the number of characters encoded, starting from the offset. Is the offset is set to zero, it means the dictionary will not be used to encode the next symbol. In this case, the length receives the literal value that correspond to the unmatched symbol. The pairs are further compressed using two separate huffman trees, one for the offset and the other for the length.

Figure 6 shows what codes are generated for the running example message. The circle marks the current symbol. Symbols before the current symbol are part of the dictionary. Non-delimiter symbols are encoded as literals (no dictionary entry is used). The symbols from the first run of delimiters are also encoded as literals. Observe that the second (or the third) delimiters of this run could point to a dictionary entry, since a delimiter is already part of the dictionary. However, small sequences are not encoded using the dictionary as a way to achieve better compression. The reason is that the literal codes are more 'huffman compressible' than the dictionary codes.

Observe that the first three delimiters from the second run are packed as a single code. These run is part of the sliding window, as denoted by the rectangle surrounding the three delimiters. The last delimiter of the run could not be encoded, since there is no run in the dictionary composed by more than three delimiters.

Message									Offset	Length / Literal
$\textcircled{;}$	;	;	a	;	;	;	;	b	0	;
;	$\textcircled{;}$	;	a	;	;	;	;	b	0	;
;	;	$\textcircled{;}$	a	;	;	;	;	b	0	;
;	;	;	$\textcircled{a}$	;	;	;	;	b	0	a
;	;	;	a	$\textcircled{;}$	;	;	;	b	4	3
;	;	;	a	;	;	;	$\textcircled{;}$	b	0	;
;	;	;	a	;	;	;	;	$\textcircled{b}$	0	b

Figure 6. Encoding the running example message with LZ

This compression scheme led to the specification of the DEFLATE standard (Deutsch, 1996), which imposes an agreement that all compliant compressors should follow, such as establishing the maximum size of the sliding window (32k) and the maximum size of the lookahead buffer (258 bytes). The standard forms the foundation of some of the most used compression algorithms used nowadays, like gzip and lzip. Besides, variations of LZ are also used by columnar databases Vertica and Sybase IQ to compress low cardinality columns.

This kind of dictionary based compression is known for associating good compression ratios and response time. Also, decompression is usually much faster than compression, since decoding a sequence of symbols is as simple as reading bytes already processed (from the sliding window). With respect to column oriented data formed by several runs of the delimiter symbol, runs already processed can be used as dictionary entries for coding other runs yet to come. The impact this kind of information have on LZ based compression is detailed in Section 4.

4. Experimental Results

The purpose of this Section is to evaluate how simple compression algorithms behave when dealing with very sparse datasets from columnar databases. Three simple algorithms were developed as part of this work. Their description is given below:

ARIT: An entropy based arithmetic encoding method that encodes a symbol based on its probability of occurrence.

RLE1: A method that encodes a run of delimiters using two bytes. The first byte is the escape code, which is the delimiter by itself. The second byte is a decimal value that represents the length of the run. Runs longer than 256 characters need to be compressed with successive codes.

RLE2: A method that encodes a run of delimiters using three bytes. The first byte is the escape code, which is the delimiter by itself. The other two bytes form a decimal value that represents the length of the run. Runs longer than 65536 characters need to be compressed with successive codes.

To perform a meaningful evaluation, we compared these methods with the commercially available GZIP, representing a method from the LZ family. All algorithms were written in C and no compiler optimization flags were set. Also, we observe that GZIP could be fine tuned to spend more time looking for matches inside the sliding window, trading a better compression ratio for a worst execution time. The default is to let GZIP decide when to stop looking for better matches (longer sequences). It turns out that the default setting is the one that usually brings the best results. Therefore, no fine tuning parameters were set either.

We used as dataset a database with tables generated from TPC-H, a well known benchmark for database performance evaluations (Council, 2008). Each column of the dataset was stored as a separate file and a special symbol was used as the field delimiter. During compression, each file was further divided into 8kb chunks. The chunks are used to emulate the way databases arrange data into pages (tranfer units between disk and memory). This size is a typical choice that databases use to reduce the amount of pages IO for random access and to allow the storage of reasonably large rows inside a page. For the sake of comparison, MySQL uses 16kb as the default page size and Oracle 10g uses pages from 4kb to 8kb. Also, the chunks where individually compressed. The reason is simple: limiting compression to single pages allows random access to a page without having to decompress the whole file.

We report the results achieved when compressing the following three columns from the CUSTOMER table.

- **COMMENT:** a column that stores user comments as natural text with length ranging from 29 to 116 characters.
- **ACCBAL:** a column that stores account balances with a two digits precision. Examples: '121.65' and '9561.95'.
- **MKTSEGMENT (MKT):** a column that stores one of five values of market segments. Examples: 'BUILDING' and 'HOUSEHOLD'. We artificially reduced the size of the file by running a preprocessing step that replaced values with a single character.

These columns are representative samples from three groups. The COMMENT column represent columns whose values are reasonably long. The MKT column represent columns whose values are single characters. The ACCBAL column represent columns whose values lie in-between these two extremes. Naturally, when values are shorter, the proportion of delimiters with respect to any other symbol is higher. Our intention is to verify how the compression methods behave when this proportion changes.

The selected columns were originally dense (all fields had content). Since our purpose is to evaluate sparse datasets, we artificially change the datasets by replacing fields at random positions with empty values[1]. The testbed is composed by several very sparse datasets where the amount of empty values varies from 99.05% to 99.95% of the total number of fields.

Figure 7 shows the COMMENT column results. On the left we present the compression ratio in terms of bits per code (bpc) achieved when varying the sparsity. GZIP is clearly the winner, regardless of the sparsity considered. The two versions of RLE behave similarly, and gradually approach the compression of GZIP. However, they are no match

[1]Null values could also be easily be accounted, if we assume a presence vector is used inside the pages

for GZIP even when more than 99.9% of the values are empty. Besides, we can see that ARIT is better than RLE when the sparsity is reduced. However, it is still much worst than GZIP.

On the right of Figure 7 we present the time needed to compress all chunks and the time needed to decompress them. We report the results when using the highest level of sparsity (99.95%). The first thing to notice is that ARIT is slower than the alternatives in compression and decompression. On the other hand, there is no significant difference in execution time considering GZIP and the run length methods.

Another interesting fact is that decompression is generally slower than compression. This behaviour was observed even with GZIP, where decompression is usually faster. We have also observed that the speed oscillates when working with different levels of sparsity. This lack of linearity indicates that the processing cost of GZIP and RLE is not a driving factor when working with small sized problems. For instance, with a 99.95% sparsity, the compression of GZIP uses 0,09 bpc. Chunks with 8kb becomes compressed pages of approximately 92 bytes. The decompression cost in this case is clearly affected by the fluctuations that occur during IO operations or CPU usage.

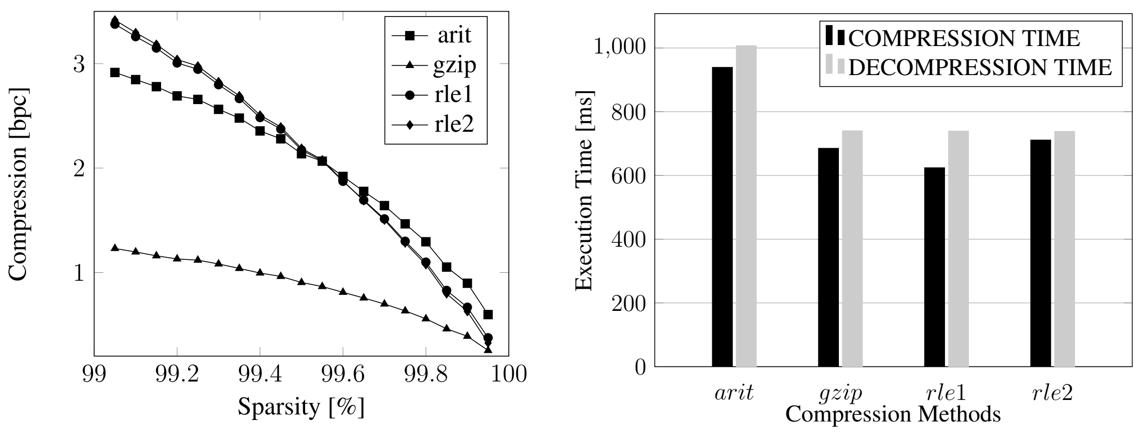


Figure 7. Compression results of CUSTOMER.COMMENT

Figure 8 shows the ACCBAL column results (compression in terms of bits per code on the left and execution time with 99.95% sparsity on the right). The first thing to notice is that the compression of ACCBAL is generally better than COMMENT. The reason is rather obvious: the information entropy is reduced when the frequency of the delimiter increases and the other symbols become less meaningful. In other word, patterns become more frequent than other arbitrary sequences, and compression methods are build to leverage from patterns.

It is also interesting to observe that the RLE versions were able to achieve much better results than before. For once, they are better than ARIT regardless of the sparsity considered. Additionally, RLE2 overpasses GZIP when sparsity reaches a certain level (99.8%). At this point the frequency of the delimiter reaches 98% of all symbol occurrences. For the sake of comparison, when compressing COMMENT, a sparsity of 99.8% corresponds to a delimiter frequency of 87%.

As for the execution time, the arithmetic encoding again shows the worst performance whereas the other methods had similar results. The measurements again suggest that these algorithms does not differ significantly when working with small sized problems.

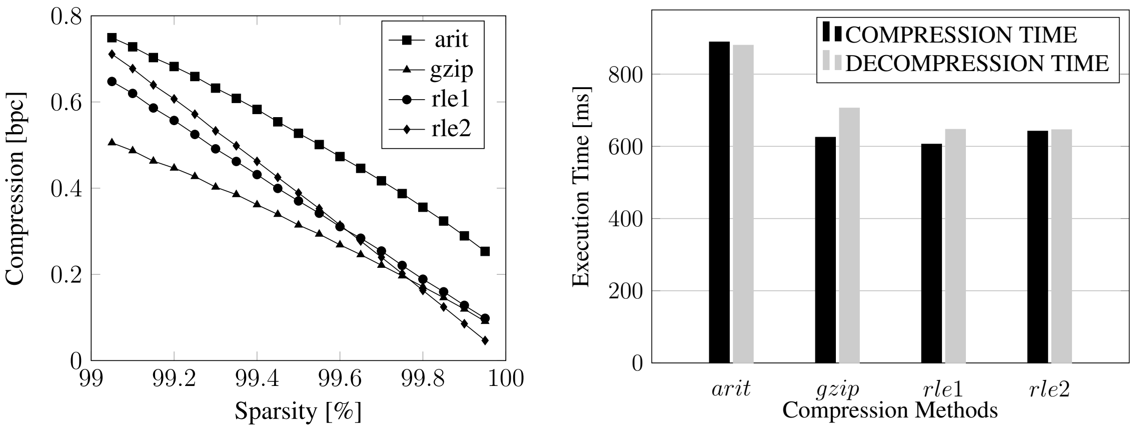


Figure 8. Compression results of CUSTOMER.ACCBAL

Figure 9 shows the market segment results. It is the test case where the delimiter appears with the highest frequency possible. Our evaluation shows how the algorithms behave with this extreme scenario. The first thing to notice is that the compression is generally higher when compared to the other tested scenarios. This happens mostly because the vocabulary is reduced (one delimiter plus five other characters), which leads to the occurrence of more patterns.

Most importantly, we call the attention to the fact that RLE2 becomes better than LZ when sparsity is above 99.3%, whereas in the previous experiment it only happened when sparsity was over 99.8%. With respect to the comparison between RLE1 and RLE2, the turning point (when RLE2 becomes better than RLE1) happens when sparsity reaches 99.65%, regardless of the column. Since these methods only compress the delimiter, the average number of characters per field does not change the turning point.

The results on the right side of Figure 9(running time with 99.95% sparsity) satisfy our previous observations that all but the arithmetic encoding method have a similar running speed when the message is small. This is interesting remark, if we consider that the key strength of GZIP is speed. When sparsity is high and pages are small, other methods become equally competitive.

5. Concluding Remarks

We have shown that, when sparsity is very high, column oriented data that is normally compressed using variations of the LZ method may also benefit from simpler methods based on the run length of delimiters, both in terms of compression ratio and speed.

With respect to execution time, the cost for compression and decompression of the RLE methods evaluated are similar to LZ when the files are segmented as pages. With respect to compression ratio, the results indicate two things. First, only sparsity matters if one wants to decide between the two RLE versions presented. However, if one wants to decide between RLE and GZIP, the average field size also matters. For columns where the fields are long, the sparsity would have to be much higher in order for RLE to overcome LZ.

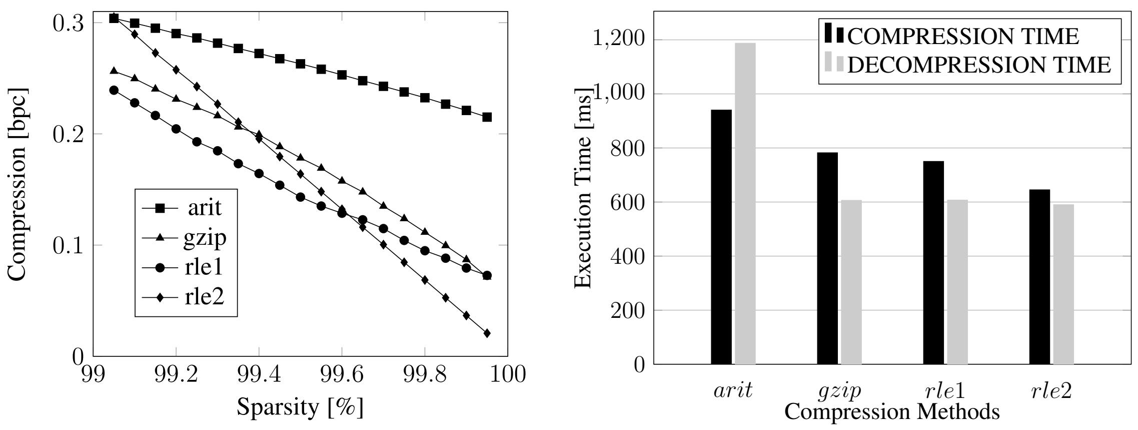


Figure 9. Compression results of CUSTOMER.MKTSEGMENT

One interesting (and useful) property of RLE is that it enables direct computation of counts of undefined values. However, unlike other lightweight methods (applied to low cardinality values), it does not provide direct access to random values. One possible way to overcome this limitation is the usage of page layouts designed to handle very sparse datasets composed by high cardinality values. This idea is similar in spirit with RLE, in the sense that no characters are encoded (other than the delimiter), and we have shown that this strategy is promising in some scenarios. The investigation of this novel page layout is left as future work.

References

Abadi, D., Madden, S., e Ferreira, M. (2006). Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 671–682. ACM.

Abadi, D. J. et al. (2007). Column stores for wide and sparse data. In *CIDR*, pages 292–297.

Ailamaki, A., DeWitt, D. J., Hill, M. D., e Skounakis, M. (2001). Weaving relations for cache performance. In *Proceedings of the 27th International Conference on Very Large Data Bases*, VLDB '01, pages 169–180, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Burrows, M. e Wheeler, D. J. (1994). A block-sorting lossless data compression algorithm (technical report).

Council, T. P. P. (2008). Tpc-h benchmark specification. [Online; accessado em 19 de janeiro de 2016].

Deutsch, L. P. (1996). Deflate compressed data format specification version 1.3.

Huffman, D. A. et al. (1952). A method for the construction of minimum redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101.

Lamb, A., Fuller, M., Varadarajan, R., Tran, N., Vandiver, B., Doshi, L., e Bear, C. (2012). The vertica analytic database: C-store 7 years later. *Proc. VLDB Endow.*, 5(12):1790–1801.

Matei, G. e Bank, R. C. (2010). Column-oriented databases, an alternative for analytical environment. *Database Systems Journal*, 1(2):3–16.

Skodras, A., Christopoulos, C., e Ebrahimi, T. (2001). The jpeg 2000 still image compression standard. *IEEE Signal processing magazine*, 18(5):36–58.

Witten, I. H., Neal, R. M., e Cleary, J. G. (1987). Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540.

Ziv, J. e Lempel, A. (1978). Compression of individual sequences via variable-rate coding. *Information Theory, IEEE Transactions on*, 24(5):530–536.

Zukowski, M., Heman, S., Nes, N., e Boncz, P. (2006). Super-scalar ram-cpu cache compression. In *Proceedings of the 22Nd International Conference on Data Engineering*, ICDE '06, pages 59–, Washington, DC, USA. IEEE Computer Society.