Computing and Informatics, Vol. 37, 2018, 405–423, doi: 10.4149/cai_2018_2_405

COMPRESSION OF TEXTUAL COLUMN-ORIENTED DATA

Vinicius Fulber Garcia, Sergio Luis Sardi Mergen

Department of Languages and Computer Systems
Universidade Federal de Santa Maria
Av. Roraima, 1000, Santa Maria, Brasil
e-mail: {vfulber, mergen}@inf.ufsm.br

Abstract. Column-oriented data are well suited for compression. Since values of the same column are stored contiguously on disk, the information entropy is lower if compared to the physical data organization of conventional databases. There are many useful light-weight compression techniques targeted at specific data types and domains, like integers and small lists of distinct values, respectively. However, compression of textual values formed by skewed and high-cardinality words is usually restricted to variations of the LZ compression algorithm. So far there are no empirical evaluations that verify how other sophisticated compression methods address columnar data that store text. In this paper we shed a light on this subject by revisiting concepts of those algorithms. We also analyse how they behave in terms of compression and speed when dealing with textual columns where values appear in adjacent positions.

Keywords: Compression, column-oriented databases, LZ, PPM, BWT, entropy encoding, DSM, NSM, PAX

Mathematics Subject Classification 2010: 68P30

1 INTRODUCTION

Traditional relational databases use a page layout called NSM (N-ary Storage Model) where rows are stored contiguously on disk. Recent works propose a different page layout called PAX (Partition Attribute Across) where columns of relational tables

are stored contiguously on disk. A similar physical organization is also employed by column-oriented databases, such as MonetDB.

This innovative data arrangement provides faster access for specific query patterns, such as those requiring a small amount of columns. The reason is that less IO is needed to retrieve the values of interest, as they fit in less data pages – no space is wasted in the page with values from columns that are not needed [12]. Additionally, all information stored in a page belong to the same domain and data type. This homogeneity allows data compression algorithms to achieve higher compression ratios if compared to physical arrangements like NSM where data inside a page is much more diverse [1].

Compression in column-oriented data is achieved in several different ways, depending on the nature of data. Sybase IQ [14] and Vertica [11], two of the industry's leading column-oriented databases, use variations of the LZ compression method when compressing high-cardinality texts. LZ is suited for data with a certain degree of redundancy, such as sentences written in natural language, where words are grammatically linked. The method is able to encode redundant parts effectively, achieving good compression associated with a low execution time.

We note that there are other compression methods suited for texts, such as PPM and BWT. Several works (e.g. [9]) report empirical results achieved when applying these methods on the Calgary Corpus, a popular collection of documents used for compression [4]. However, the redundancy in column-oriented data is naturally different than redundancy found in conventional files. Also, there is a limit on the size of a database page. The question of how such methods behave when compressing columnar text organized as pages still deserves investigation.

The goal of this paper is to compare how effective are the BWT, LZ and PPM methods with respect to compression and execution time when compressing column-oriented textual data. We start describing our motivation and running example (Section 2). From Section 3 to 6 we revisit the concepts behind the methods that are part of the evaluation. For each method we outline in general terms how the patterns found are explored to achieve compression. In the final part the paper is dedicated to reporting on the experimental results (Section 6) and presenting our concluding remarks (Section 7).

2 MOTIVATION AND RUNNING EXAMPLE

Figure 1 shows different page layouts for a table containing columns YEAR, STATUS and COMMENT. The NSM is the typical design choice of relational databases that store rows contiguously on disk. Conversely, DSM (Decomposition Storage Model) and PAX are designed to keep values of the same column together [2]. DSM splits a table into as many columns as it has. Then, each column is stored in a separate group of pages. PAX follows the same principle, but uses the concept of mini-pages inside a page. The rationale is that whole records can be read from a single PAX page. DSM is the precursor of modern columnar databases, while the general idea of

PAX is employed by some relational database vendors, like the the Hybrid Columnar Compression (HCC) used by Oracle Exadata [3].

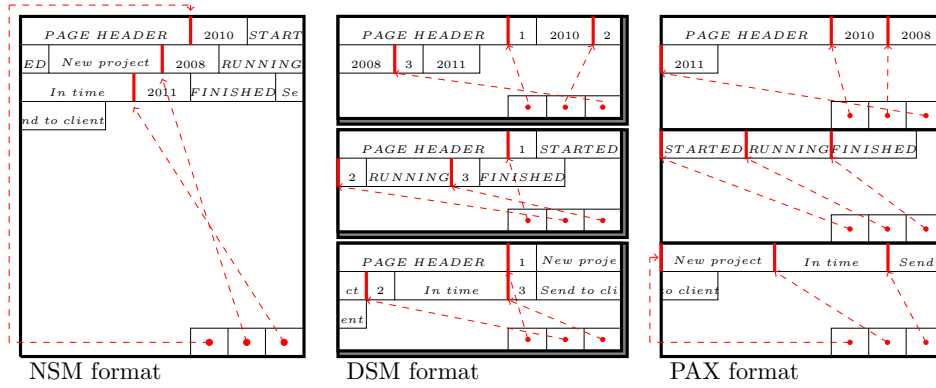


Figure 1. Three different page layouts

In what follows, we make a distinction between heavy-weight and light-weight compression methods. We use the term heavy-weight to refer to methods where the encoding of upcoming symbols relies on information gathered from the previously encoded symbols. The term light-weight is used otherwise.

The first thing to notice about the illustrated example is that the values of the STATUS and YEAR columns in the DSM/PAX page layouts are particularly suited for light-weight compression methods, such as dictionary encoding and frame of reference, respectively. In the former, one value is coded as an index that points to a dictionary where all possible status values can be found. In the latter, one value is coded as the difference from a reference value (the average stored year value).

Some light-weight methods (like the ones mentioned above) compress values into fixed-width codes. This allows fine grained decompression of single values, without the need of full-page decompression. There is also the possibility to operate directly on compressed data. Working with compressed data means that more information fits in memory and the number of cache misses is reduced if the same value is needed again. Also, the vectorized compressed data can be more efficiently handled by modern CPUs that can pipeline and parallelize instructions [19].

The shortcoming of light-weight compression methods is that they do not address well skewed and high-cardinality values, such as the ones found in the COMMENT column. Values of comments are typically sentences from a written language. In this case, the patterns that emerge are different, comprising root words, frequent sequences of contiguous words, or even a high frequency of single words, such as prepositions and nouns. Heavy-weight methods are best suited under these circumstances.

When using a heavy-weight method, the execution engine of a query processor cannot operate on the compressed data directly. The whole page/mini-page needs to be decompressed before a specific value can be accessed. The need for decompressing text obviously slows down query execution. On the other hand, the IO cost is reduced. For instance, leaving COMMENT uncompressed (or poorly compressed) in the PAX format may hinder the benefits of using light-weight methods to compress other mini-pages, resulting in less rows per page and more data transfers. Besides, a mini-page storing text occupies more space than a mini-page storing data types typically supported by light-weight methods, which makes text compression even more critical.

This is the motive that drove us into investigating compression methods suited for high-cardinality and skewed textual values. Throughout the remaining of the paper we revisit concepts of well-known heavy-weight compression methods in order to understand how different they are and why they are strong candidates to compress this sort of data.

Before we begin, we call the attention to the pages depicted in Figure 1 and the fact that data grow from the one side and auxiliary vectors grow from the other side. The vectors are useful for indirect access inside a page/mini-page. Examples are presence vectors to indicate nullable fields and offset vectors to indicate where a variable length record (in the case of a NSM) or a variable length field (in the case of PAX/DSM) begins.

Observe that the usage of heavy-weight compression methods implies that random access is not possible. Therefore, there is no need for an offset vector. We argue that, in those cases, the page/mini-page design can be simplified by dropping this kind of bookkeeping. If a column accepts texts with variable length, instead of storing offsets, a special delimiter symbol could be used to separate one value from the next.

For instance, suppose there are seven fields of the COMMENT column. The value of the forth field is 'abc' and the other fields are either nulls or blanks. Using the semicolon (;) as the special delimiter symbol, the fields put together become as follows:

;	;	;	a	b	c	;	;	;

This is the information to be encoded, and the one we use as the running example from now on. Observe the presence of two runs, for leading and trailing delimiters. This is a kind of pattern that occurs when values of a column appear in adjacent positions. It is a simple example, but it serves our purpose of illustrating how patterns are coded by the investigated methods.

Throughout the rest of the paper we use the term *message* to refer to the contents of the running example. We also use the term *symbol* to indicate each byte of the *message*. We assume the files use the ASCII encoding scheme, so that each byte maps to a different *symbol*. Compression is measured as bits per code (bpc), the average number of bits needed to code a character.

3 ENTROPY ENCODING

In information theory, the entropy of a *message* is a measure of how unpredictable the *message* is. A higher entropy means it is more difficult to predict what *symbols* are more likely to appear. For instance, *messages* where one *symbol* is much more common than the others (like the delimiter in a very sparse dataset) have a low entropy, since we can predict that *symbol* will appear quite often.

The purpose of entropy encoding is to approximate the entropy of a *message*, that is, to use the minimal amount of bits to represent information. The lower the entropy, the more compressed the *message* can be. Two of the most popular compression methods based on entropy are the Huffman coding [10] and arithmetic coding [16]. In what follows we present the differences between them.

Huffman Coding: The Huffman coding assigns to each *symbol* of the *message* a unique and unambiguous sequence of bits, reserving the smaller sequences to the most frequent *symbols*. A binary tree can be used to create the mapping through a greedy algorithm that iteratively puts the two nodes (*symbols*) with the minimum frequencies under the same parent. Figure 2 shows the tree generated based on the contents of the COMMENT *message*. Observe that coding the delimiter requires a single bit. After compression, the nine characters are transformed into a 15 bit sequence (000101101110000), giving a compression of 1.66 bpc.

Symbol Mapping

Symbol	Frequency	bit sequence
;	6	0
a	1	10
b	1	110
c	1	111

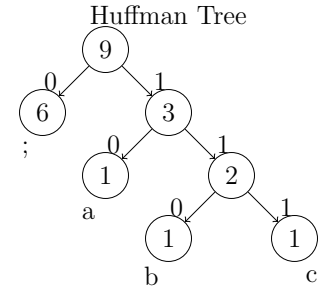


Figure 2. The Huffman tree created based on the comment *message*

Arithmetic Coding: Unlike the Huffman code, there is no unique bit sequence that determines each *symbol* in the arithmetic coding method. 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 architectural 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 3 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 66 %. 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 drops to 43 %. The probability keeps being updated as the remaining *symbols* are processed, and eventually the most significant bits are flushed.

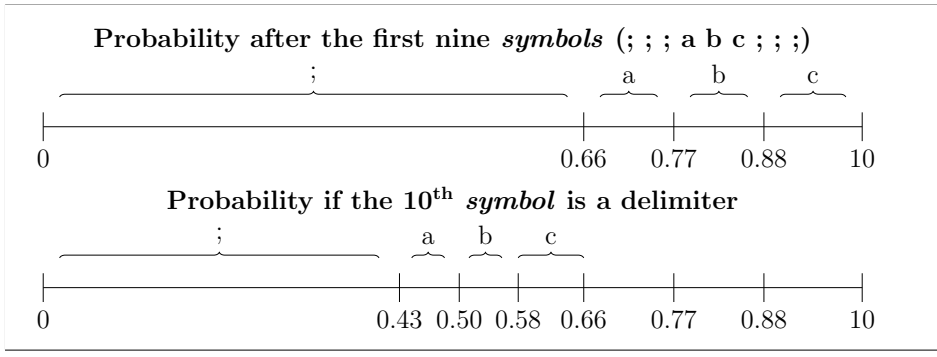


Figure 3. Encoding probabilities based on the running example *message*

In comparison to the 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, instead of having a fixed precomputed probability. On the other hand, Huffman codes allow reading from arbitrary positions of the compressed *message*, which is not possible using arithmetic coding. In either case, the effectiveness of both methods is highly dependent on the existence of very frequent *symbols*. In most scenarios the entropy coding is not used alone, but as part of a more complex method, such as BWT, LZ and PPM, as we detail next.

4 BURROWS WHEELER TRANSFORM (BWT)

The compression method proposed by [5] is divided in stages, as illustrated in Figure 4. The information flows from left to right and each stage transforms data into a format suited to the next stage.

Burrows Wheeler Transformation Stage: During the first stage, the *message* (with n characters) is copied into n rows, where row i is the same as row $i - 1$

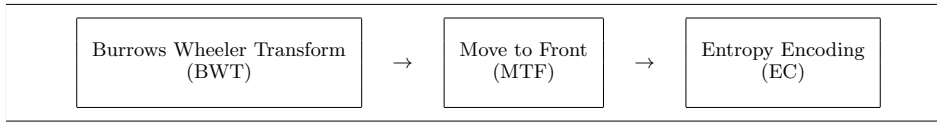


Figure 4. BWT Stages

rotated one character to the right and row 0 is the original *message*. The rows are then ordered lexicographically. From this arrangement, two data elements are passed to the next stage. The first is the content of the last column of the sorted rows. The other is the index of the sorted rows that contains the original *message*. The purpose of the latter is to reconstruct the *message* during decompression.

The reasoning behind BWT is to explore the fact that some characters usually come before other characters in many written languages. In such cases, a character that usually precedes others tends to appear in adjacent positions in the last column. As we discuss later, this is desired when it comes to compression. To illustrate, Figure 5 presents the Burrows Wheeler transformation of the COMMENT *message*. Observe that all delimiter *symbols* appear next to each other in the last column of the sorted matrix. Curiously (and mostly because the *message* is small) the leading and trailing runs of ';' were merged into a single run.

1	;	;	;	a	b	c	;	;	;		;	;	;	;	;	;	a	b	c
2	;	;	;	;	a	b	c	;	;		;	;	;	;	;	a	b	c	;
3	;	;	;	;	;	a	b	c	;		;	;	;	;	a	b	c	;	;
4	;	;	;	;	;	;	a	b	c		;	;	;	a	b	c	;	;	;
5	c	;	;	;	;	;	;	a	b	→	;	;	a	b	c	;	;	;	;
6	b	c	;	;	;	;	;	;	a		;	a	b	c	;	;	;	;	;
7	a	b	c	;	;	;	;	;	;		a	b	c	;	;	;	;	;	;
8	;	a	b	c	;	;	;	;	;		b	c	;	;	;	;	;	;	a
9	;	;	a	b	c	;	;	;	;		c	;	;	;	;	;	;	a	b

Before the sort operation — After the sort operation

Figure 5. Applying the Burrows Wheeler transform to the comment *message*

Move To Front Stage: The purpose of the MTF stage is to encode each character of the last column as a numeric index ranging from zero to 255. This index refers to a lookup table that contains all possible characters. The table is first built with characters occupying arbitrary positions. Then, when a character is encoded, it is moved to the front of the list. When the next character to encode is the same as the previous one, the resulting index position is zero, since the character looked up is now found in the first position of the table. At the end of the transformation, the result is expected to be formed by many consecutive zeros. Also, few index positions will be very frequent (the smaller ones). This allows entropy coding to be performed, as we detail next.

Figure 6 presents the output generated after each stage of BWT (for the running example). Line 3 shows how MTF transforms the *symbols* received from the previous stage. To get this result we assume that the characters ';', 'a', 'b', 'c' initially occupy the positions 0, 1, 2 and 3 of the lookup table, respectively. Observe that every repeated adjacent *symbol* is encoded as zero.

1. original *message*	;	;	;	a	b	c	;	;	;
2. BWT output	c	;	;	;	;	;	;	a	b
3. MTF *symbols*	3	1	0	0	0	0	0	2	3
4. EC output	1110	10	0	0	0	0	0	110	1110

Figure 6. Encoding the running example *message* with BWT

Entropy Encoding Stage: The input for this stage is composed by a few indexed positions with a very high frequency. Entropy coders (like Huffman and arithmetic encoding) can explore this property to actually achieve compression. With respect to our running example, the Huffman tree created for the indexes would yield the sequences '0', '10', '110' and '1110' for the values 0, 1, 2 and 3, respectively[1]. Line 4 of Figure 6 shows the Huffman coding of the input received from the MTF stage. In this particular case the compressed data is three bits longer than the data compressed when only the Huffman encoding is used (Section 3). This is mostly due to the size of the example. As we demonstrate on the experimental section, the behaviour is different when dealing with larger files.

Decompression is achieved by executing the stages in opposite direction, starting from the entropy encoding and finishing with the Burrows Wheeler Transformation. All stages are reversible, including the transformation. The last column along with the index of the original text is enough information to reconstruct the original *message*.

The BWT method is effective in compressing text, especially if data is domain specific. The more specific is the domain, the shorter is the set of *symbols* that precedes characters. This translates into longer runs of the same *symbol* after the BWT stage. Runs of the delimiter *symbol* may also translate into runs of the same delimiter, as the example shows. A handicap of BWT is that it is very memory intensive, since the matrix transformation requires the whole *message* to be read. To reduce memory requirements, the input is divided into blocks, and each block is compressed separately. Higher compression ratios can be obtained when working with longer blocks, as we demonstrate in Section 7.

[1] The Huffman codes (or *symbols* frequencies) need also be saved as part of the encoded file to allow decompression.

5 LEMPEL-ZIV

This compression method compresses sequences of *symbols* of varying sizes by replacing them with a code that refers to a dictionary entry. The dictionary is formed by *symbols* of the *message* that were already processed.

The idea of using the previous *symbols* as a dictionary was originally proposed by Lempel and Ziv, which is why this kind of dictionary-based compression method is commonly referred to as Lempel-Ziv, or LZ to short. The name LZ77 is used to identify the original idea, where the dictionary is a sliding window formed by past *symbols* [17]. The LZ78 is a variant where the dictionary is explicitly built as a table, and its entries are accessed by an index [18].

Several other variations appeared, such as LZW (used in Sybase IQ) and LZO (used in Vertica). The one we have presented here is based on LZ77. It is a simplification of the method used in GZIP, where the code is formed by a pair (distance, length). The distance is an index to a position of the dictionary where a sequence starting at the current *symbol* is found. The length is the amount of *symbols* to be coded from that index position. The distance is incremented backwards from the current *symbol*. The value zero indicates that the current *symbol* was not found in the dictionary. In such cases the length is replaced by the actual *symbol*.

Figure 7 shows what codes are generated for the COMMENT *message*. The current *symbol* is circled. Symbols before the current one becomes the part of the sliding window. The first six *symbols* are encoded as literals (no dictionary entry was used). The last three *symbols* (a sequence formed by repeated delimiters) are packed as a single code. Observe that the second (or the third) *symbol* of the *message* could also point to a dictionary entry, since a delimiter is already a part of the dictionary. However, coding small sequences instead of outputting literals may actually result in higher codes.

Message									Distance	Length/Literal
ⓘ	;	;	a	b	c	;	;	;	0	;
;	ⓘ	;	a	b	c	;	;	;	0	;
;	;	ⓘ	a	b	c	;	;	;	0	;
;	;	;	ⓐ	b	c	;	;	;	0	a
;	;	;	a	ⓑ	c	;	;	;	0	b
;	;	;	a	b	ⓒ	;	;	;	0	c
;	;	;	a	b	c	ⓘ	;	;	6	3

Figure 7. Encoding the comment *message* with LZ

Since the codes refer to a part of the *message* that have already been processed, during decompression it is possible to use the codes in order to reconstruct the original *message*. Decompression is much faster than compression, since there is

no need to locate *symbols* from the sliding window. Instead, it only needs to copy *symbols* from the sliding window (based on the code) into the output.

GZIP implements the DEFLATE standard [7], which imposes an agreement on the code format, the maximum size of the window (32 k), the maximum length ahead (258 bytes), among others. It also establishes that the distances and lengths are further compressed using two separate Huffman trees.

Compression tools that implement this standard (like GZIP) are heavily used. One reason for the popularity (apart for being a recommendation) is that it is relatively straightforward to implement a decompression algorithm that is compliant with the standard. Besides, this kind of dictionary-based compression is not only fast but also achieves good compression ratios for most of the file formats.

With respect to column-oriented textual data, the occurrence of many similar (or equal) values can also be encoded as pointers to a dictionary entry. Common expressions already processed can be used as dictionary entries for coding occurrences of other common expressions yet to come. The impact this kind of information has on a LZ based compression method is detailed in Section 7.

6 PPM

The PPM (Prediction by Partial Matching) compression method codes one *symbol* at a time. Given the *symbol* to code, PPM predicts the probability of occurrence of that *symbol*. This value is then coded using arithmetic encoding. Highly frequent *symbols* are encoded with fewer bits. The main idea can also be adapted so that other entropy coders (such as Huffman) can be used.

The probability estimation takes into account the context, which means the set of *symbols* that precedes the *symbol* being coded. Most PPM approaches use Markov models of different orders to issue a prediction. If the highest-order model is unable to predict a *symbol* (it never occurred in that context before), the next-higher-order model is used. In the worst case this goes on until reaching the zero-order model that is able to predict all *symbols*.

When a *symbol* cannot be predicted by an order, PPM issues a signal indicating that a *symbol* never seen before has appeared. This special signal is also referred to as escape *symbol*. The probability of the escape depends on the variant of the PPM used. One of the proposed ideas (called PPM-C) was to count the number of different *symbols* that occurred in that context and use it to compute the probability [13].

To illustrate, consider the COMMENT *message* reintroduced below. The arrow points at the next *symbol* to encode and the curly bracket indicates the context to be used to determine the probability of the next *symbol*. The length of the context corresponds to the length of the highest Markov model. Empirical results reported by [13] show that compression is best when using at least four as the higher order. For the sake of presentation we use a maximum order of two.

; ; ; a b c ; ; ; ?
Context

Figure 8 shows the generated context tree. A context in this tree is formed by the *symbol* of a node concatenated with all of its parents. The parenthesis indicates the frequency of that context. For instance, the leftmost leaf node shows that the context ';;;' has already occurred twice. The circles indicate the current contexts from the highest to the lowest order.

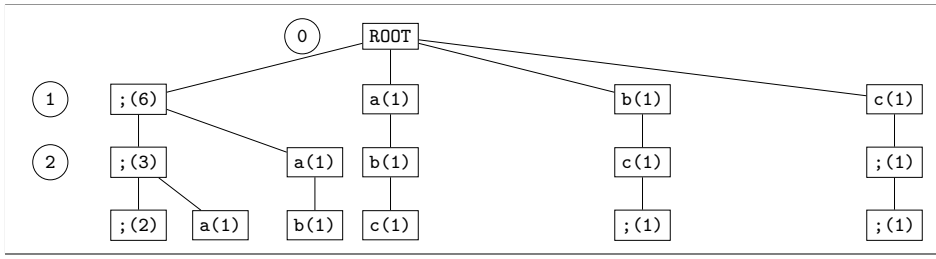


Figure 8. Context tree of the comment *message* (maximum order of two)

If the next *symbol* to encode (at the arrow position) was a delimiter, the highest order 2 (';;') would issue a probability of 40 % (two delimiters out of five occurrences: the delimiters themselves, one character 'a' and two escapes). If the *symbol* was the character 'a', the probability would be lower (20 %). If the *symbol* is new in that context, like the character 'b', an escape probability is issued (40 %) and the search continues in the order 1 (';'). Symbols with equal probability do not conflict since they occupy different intervals in the probability scale.

One technique that achieves better compression is to ignore *symbols* that exist in higher orders. For instance, if the incoming *symbol* is 'b', it would be found at order 1. In this case, the 'b' probability at order 1 would be 25 % (one out of four), if *symbols* from the upper level are ignored, and 7 % (one out of thirteen) otherwise.

The compression process is reversible. The encoding uses the context model, which is constructed based on the *symbols* already processed. Thus, given a probability, it is possible to use the current model constructed so far in order to locate the next *symbol* to be decoded.

This technique is very useful for compressing texts written in natural language, since it is able to predict with a high level of hit rate the next character of small words (or roots/prefixes/suffixes) where their length is below the maximum order. Given this, it is usually better than LZ to code small patterns. Also, it is able to encode patterns found outside the LZ window. The bottleneck is decompression speed, since PPM needs to traverse the list of children of a node to find out the one to decode. The cost can be amortized by keeping the children sorter by frequency, but the cost is still meaningful, as we discuss next.

7 EXPERIMENTAL RESULTS

The purpose of this section is to evaluate how the three investigated compression methods behave with columnar data. The experiments are divided in three parts. First we analyse how the compression varies according to the nature of textual data. Then we show how the compression and compression/decompression speed varies according to the amount of data compressed. Finally we evaluated alternative LZ based methods.

Initially, two commercially available compression tools were evaluated: BZIP2 (version 1.0.6) and GZIP (version 1.2.4) that implement the BWT and LZ methods, respectively. To avoid biased evaluations, the executables were used as is, without any kind of tuning, and the meta-data overhead was stripped from the compressed output. We also evaluated an implementation of PPM-C, implemented as part of this work. Our version uses four levels of context and it ignores symbols from higher orders in order to improve compression. All algorithms were written in C.

The algorithms were tested on a Pentium Dual-Core with 2.5 GHz. Compression speed was measured as the amount of data that is compressed by time, and decompression speed is measured as the amount of data that is decompressed by time. The time obtained is an average of 30 executions, ignoring the 10 % lesser and higher times. The experiments were held in a minimalist operating system, where functions are reduced to a bare minimum that does not sacrifice stability.

7.1 How the Data Format Impacts Compression

Throughout the paper we have seen that all of the investigated compression methods are able to handle textual data, and they explore the patterns found in the already processed *message* to achieve compression. However, text can be organized in very distinct ways, following a rigid or relaxed structure, or having no structure at all. Here we analyse how compression is affected by different text formats.

Two datasets were used, one for columnar data and the other for conventional files. The columns were taken from the TPC-H benchmark, which is a set of tables commonly used to evaluate database transaction processing features [15]. The conventional files were taken from the Calgary Corpus. As mentioned earlier, it is a set of files traditionally used to evaluate compression algorithms.

Results are presented for three high-cardinality columns from TPC-H: CUSTOMER.COMMENT, LINEITEM.COMMENT and PART.NAME. Each column was stored as a separate file, and a reserved *symbol* was used to separate one value from the next. The columnar data are compared against four files from the Calgary Corpus: a book (BOOK2), a paper (PAPER2), a source code written in Pascal (PROGP), and a list of bibliographic references (BIB). Other files were considered as well (from TPC-H and Calgary), and the variations observed were the same.

All uncompressed files were divided into chunks of 16 KB, and each chunk was compressed separately. What we report is the average compression achieved for each file, measured as bits per code (bpc). The compression is limited to chunks (and

not the whole file) to emulate database compression schemes that handle one page at a time.

Figure 9 shows how the compression varies according to the nature of textual data. The results show that BZIP2 and PPM achieve better compression than GZIP in most cases. Additionally, the columnar values are more compressible in comparison to other texts written in natural language and even to the well behaved PROGP. One of the possible reasons is that it is more likely to find patterns in a list composed by small textual fields than it is in a single and longer instance of a text.

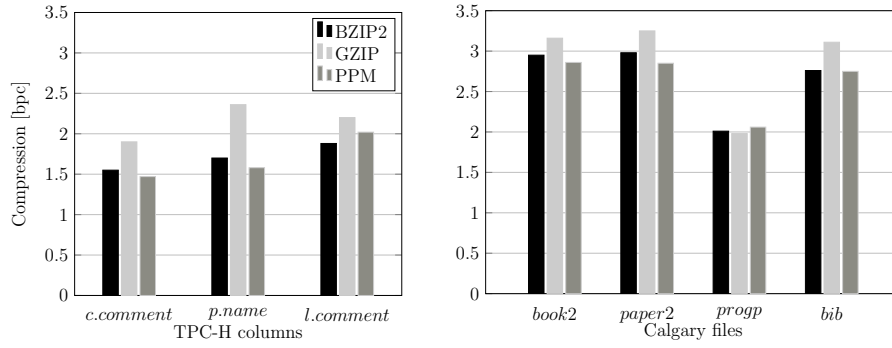


Figure 9. Compression results

Interestingly, the compression improvement of BZIP2 and PPM over GZIP is more meaningful with the columnar values. For instance, PPM uses almost 33 % less bits than GZIP when compressing PART.NAME. The improvement of PPM over GZIP with the Calgary files is less apparent. With the highly structured PROPG, PPM is actually worst. This behavior lead us to the conclusion that BZIP2 and PPM explore higher levels of redundancy better than GZIP.

7.2 How the Data Length Impacts Compression

This experiment is focused on columnar data stored in database pages. It shows how the bpc and compression/decompression speed varies according to how large the pages are. The evaluation was done by compressing data chunks of different sizes. We focused on the compression of the PART.NAME column, but similar results were obtained for other textual columns as well.

The size of a chunk is related to the amount of uncompressed data, not to the size of a page. For instance, GZIP takes 1.7 bits per code to encode 16 KB of comments, resulting in 3.481 bytes of compressed text in a page of unspecified size. What we need are measures taken from the full occupation of a page with a determined size. We did this by applying a linear interpolation of the results achieved when compressing chunks of uncompressed data. The results are shown in Figure 10. Solid lines indicate compression and dotted lines indicate compression

speed (at the left) and decompression speed (at the right) as the amount of bytes coded per millisecond. The size of the pages vary from 1 KB to 128 KB.

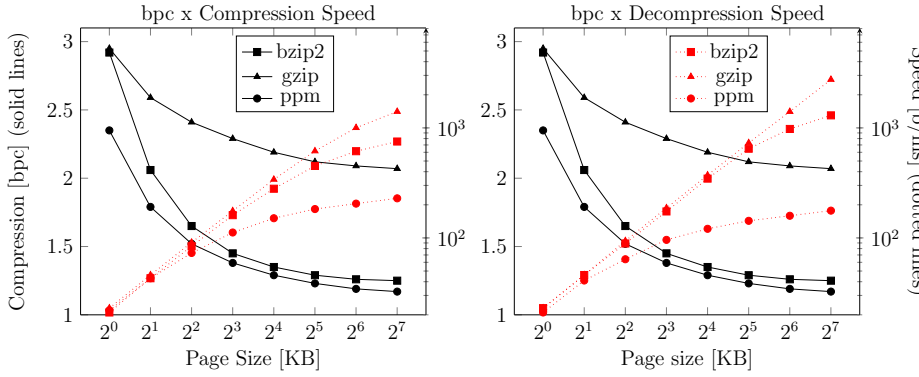


Figure 10. Measuring the compression of the name column

As the page size gets longer, GZIP becomes notably faster than BZIP2, which is in turn faster than PPM. Despite the speed, GZIP does not explore the size factor as well as the alternatives to obtain better compression. The reason is related to the 32 KB sliding window of GZIP. A longer *message* means more patterns can be found. However, if the patterns lie beyond the limits of the sliding window, they are not used to encode the incoming *symbols*.

For sufficiently long *messages*, GZIP is the clear winner. It may loose regarding the compression, but the speed compensates, especially for decompression. However, databases use pages as the transfer unit between disk and memory, and pages have a rather limited size. For the sake of comparison, MySQL uses 16 KB as the default page size and Oracle 10g uses pages from 4 KB to 8 KB. Therefore, it makes sense to consider scenarios where smaller *messages* need to be compressed. This type of scenario is analysed in the next section.

7.3 How Page-Fitting Messages Impacts Compression

As we are working with columnar compression, two page layouts are considered: DSM and PAX. These layouts subsume the main strategies adopted when values of the same column are stored together. Our intention was to evaluate the methods effectiveness when filling pages with compressed data.

When working with the DSM page layout, the whole page is dedicated to a single column. We can see from Figure 10 that BZIP2 is an interesting solution for the storage of PART.NAME values using this layout and regular page sizes (from 4 KB to 16 KB). To give a better look, Table 1 details the measurements obtained when storing small messages. BZIP2 achieves a bpc 32 % superior than GZIP when working with a 4 KB page. In contrast, GZIP is only 8 % and 6 % faster in compression and decompression speed, respectively.

When working with the PAX page layout, the page is divided into n mini-pages for the n columns it stores. Unlike DSM, only part of the page is reserved to the storage of columns whose values are textual. It means we need to look at the compression of even smaller *messages*, where the tendency is that the compression of BZIP2/PPM proportionally degrade and execution time proportionally improve, in comparison to GZIP.

To exemplify, consider an hypothetical case where the mini-page for the PART.NAME column occupies 2 KB and compare it with the storage of 4 KB of compressed values stored using the DSM layout. As Table 1 indicates, PPM is now superior than its competitors. It used 1.78 bpc to encode 2 KB of PART.NAME values, which is still almost 32 % better than GZIP. Besides, compression and decompression speeds are similar. Again, the reason is related to the sliding window of GZIP. For long *messages*, BZIP2 and PPM spend more time analysing longer parts of the *message*, whereas GZIP is bounded by the window, regardless of the *message* size. The overhead is reduced when the *message* is small, and PPM/BZIP2 become competitive in terms of execution time.

Measure	Size [KB]	bzip2 [bpc]	gzip [bpc]	ppm [bpc]
bpc	2^1	2.05	2.58	1.78
comp. [b/ms]	2^1	43	46	43
decomp. [b/ms]	2^1	46	46	41
bpc	2^2	1.64	2.40	1.52
comp. [b/ms]	2^2	84	91	73
decomp. [b/ms]	2^2	89	94	64

Table 1. Detailed measurements using pages of 2^1 and 2^2 KB

Table 2 presents some of the possible settings when 2 KB are dedicated to a single mini-page for the PART.NAME column, using PPM. For instance, if the page is 4 KB long, and each textual field occupies 40 characters, there would be enough space in the page for 230 records. For each record, 8.9 bytes of compressed data are used by the NAME field and another 8.9 bytes of compressed data are used by the fields of the remaining columns. In general, the amount of space left for other compressed columns is reasonable, especially if we consider a small amount of columns or columns that are well compressed, such as numeric values. Besides, more space for the remaining columns can be used by reducing the number of records stored or increasing the page size, as the table shows. What is worth noting here is that, if GZIP is used instead of PPM, only 68 % of the records would fit in a page, under the same conditions described in the table. If the textual values are not compressed at all, the amount is reduced to 22 %. The difference is significant, and should not be overlooked when deciding whether it is worthy to compress textual values and which compression method to use.

Page Size	Text Length	# of Records	Text Compression	Space Left
4 096	40	230	8.9	8.9
4 096	60	153	13.3	13.3
4 096	80	115	17.8	17.8
8 192	20	460	4.4	13.3
8 192	40	230	8.9	26.7

Table 2. Some settings considering PAX with a 2 KB mini-page reserved for a textual column compressed with PPM

7.4 How Other LZ Based Methods Impact Compression

The previous sections showed that GZIP is not a compelling approach for the compression of page-fitting *messages*. It is usually faster than the alternatives for large files. However, there are no remarkable differences in efficiency when files are small. Here we extend this analysis by comparing additional LZ based methods. The algorithms are presented below:

LZF (version 3.6): This method is a LZ77 variant. One of its features is that the distance/length codes produced are not further compressed using any kind of entropy encoding. Therefore, it is a fast method, especially for decompression. However, compression rate is usually compromised.

ZSTD (version 0.8.2): This is another LZ77 variant, recently proposed by Yann Collet at Facebook [6]. The entropy encoding stage is done by a fast method based on the asymmetric numeral systems (ANS) theory [8]. The author claims that ZSTD is very efficient and achieves acceptable compression.

We have also evaluated GZIP with different settings. This method can be tuned by allowing the compressor to spend more time inside the sliding window trying to find longer patterns to encode. A parameter value ranging from 1 to 9 defines how much effort should be spent. Smaller values means the search is faster, at the cost of a reduced compression. The default value (6) represents a balance between high compression and an acceptable response time. The two extremes (1 and 9) were included in the experiment, so we can verify how much compression is possible and how fast the method can be.

Table 3 brings the comparison, when compressing the PART.NAME column, considering small *messages* (that fit in a regular page) and long *messages*. There are remarkable differences between the two scenarios. The first thing to notice is that the non-LZ methods tested are not suited when *messages* are long. Despite the fact that they reach a low bpc, the processing time is much higher than the LZ alternatives. Also, GZIP-1 is an interesting choice for long *messages*. It is practically twice as more efficient that GZIP standard for compression time, and compression is only 6 % worst. Decompression time is similar, as the processing is pretty much limited to copying *symbols* from the window to the output. LZF is a little faster, as there is no entropy decoding involved.

Measure	Size [KB]	ppm	bzip2	gzip	gzip-9	gzip-1	zstd	lzf
bpc	2^1	1.78	2.05	2.58	2.58	2.74	3.01	3.73
comp. [b/ms]	2^1	43	43	46	47	47	54	49
decomp. [b/ms]	2^1	41	46	46	47	46	54	49
bpc	2^7	1.16	1.25	2.06	2.06	2.33	2.28	3.02
comp. [b/ms]	2^7	229	752	1 407	1 408	2 458	2 084	2 637
decomp. [b/ms]	2^7	177	1 300	2 751	2 776	2 781	2 326	2 912

Table 3. Comparison between lz based methods and non-lz based methods

What is worth noting is that, when *messages* are small, the compression and decompression time of all methods are levelled, and the compression achieved by PPM and BZIP2 are substantially better. In fact, PPM is particularly appealing. It is 15 % better than the second best (BZIP2) and 31 % better than the best LZ based method (GZIP). The combination of outstanding compression and competitive execution time (even for decompression) makes it a strong candidate for the compression of page-fitting columnar data.

8 CONCLUDING REMARKS

In this paper we analysed the compression of textual values from column-oriented data stored in page layouts based on DSM and PAX. We have seen that the difference in terms of number of records that fit in a page is significant when comparing the compressed and uncompressed format. This is even more important if we consider that CPUs are getting much faster compared to memory bandwidth. We argue that reducing the transfer load at the expense of having to decompress data before actually using it is an attractive trade-off.

Some of the current solutions for text compression are based on variants of the LZ method. This is a great alternative when dealing with long *messages*, since it is able to associate high compression with low execution time. However, our experiments show that other methods need to be taken into account when *messages* are short. This is the case if we consider that compression is devoted to single pages, and pages are small enough to make the BZIP2 and PPM compelling approaches.

With respect to PPM specifically, besides the applicability to text, we believe it is possible to leverage compression when dealing with short fields by adjusting the way probability is computed, especially if the fields follow a regular structure, even in the presence of outliers. For instance, dates and currency fields are expected to have the same *symbols* appearing at determined positions, and a probabilistic model is able to map this relation. This example is not only a topic we intend to further investigate, but it supports our claim that there are indeed data types that are not effectively covered by light-weight compression methods. Textual data types columns are some of them, and some others also exist.

REFERENCES

[1] Abadi, D. J.—Madden, S. R.—Hachem, N.: Column-Stores vs. Row-Stores: How Different Are They Really? Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data. ACM, 2008, pp. 967–980, doi: 10.1145/1376616.1376712.

[2] Ailamaki, A.—DeWitt, D. J.—Hill, M. D.—Skounakis, M.: Weaving Relations for Cache Performance. Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01), San Francisco, CA, USA, Morgan Kaufmann Publishers Inc., 2001, pp. 169–180.

[3] Bach, M.—Arao, K.—Colvin, A.—Hoogland, F.—Osborne, K.—Johnson, R.—Poder, T.: Hybrid Columnar Compression. Expert Oracle Exadata, Springer, 2015, pp. 67–120, doi: 10.1007/978-1-4302-6242-8_3.

[4] Bell, T. C.—Cleary, J. G.—Witten, I. H.: Text Compression. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[5] Burrows, M.—Wheeler, D. J.: A Block-Sorting Lossless Data Compression Algorithm. SRC Research Report, Palo Alto, California, 1994.

[6] Collet, Y.—Turner, C.: Smaller and Faster Data Compression with Zstandard. Retrieved from `https://code.facebook.com/posts/1658392934479273/smaller-and-faster-data-compression-with-zstandard/`, 2017.

[7] Deutsch, L. P.: DEFLATE Compressed Data Format Specification Version 1.3. 1996, doi: 10.17487/rfc1951.

[8] Duda, J.—Tahboub, K.—Gadgil, N. J.—Delp, E. J.: The Use of Asymmetric Numeral Systems as an Accurate Replacement for Huffman Coding. Picture Coding Symposium (PCS), IEEE, 2015, pp. 65–69, doi: 10.1109/PCS.2015.7170048.

[9] Effros, M.: PPM Performance with BWT Complexity: A New Method for Lossless Data Compression. Proceedings of Data Compression Conference (DCC 2000), IEEE, 2000, pp. 203–212, doi: 10.1109/DCC.2000.838160.

[10] Huffman, D. A. et al.: A Method for the Construction of Minimum Redundancy Codes. Proceedings of the IRE, Vol. 40, 1952, No. 9, pp. 1098–1101, doi: 10.1109/JRPROC.1952.273898.

[11] Lamb, A.—Fuller, M.—Varadarajan, R.—Tran, N.—Vandiver, B.—Doshi, L.—Bear, C.: The Vertica Analytic Database: C-Store 7 Years Later. Proceedings VLDB Endowment, Vol. 5, 2012, No. 12, pp. 1790–1801, doi: 10.14778/2367502.2367518.

[12] Matei, G.: Column-Oriented Databases, an Alternative for Analytical Environment. Database Systems Journal, Vol. 1, 2010, No. 2, pp. 3–16.

[13] Moffat, A.: Implementing the PPM Data Compression Scheme. IEEE Transactions on Communications, Vol. 38, 1990, No. 11, pp. 1917–1921.

[14] Moore, T.: The Sybase IQ Survival Guide. Versions 12.6 through to 15.2. Lulu.com, 1st edition, 2011.

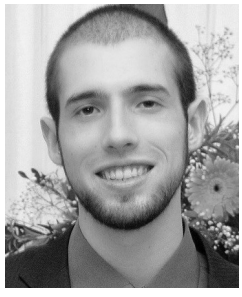
[15] Transaction Processing Performance Council. TPC-H Benchmark Specification. Retrieved from `http://www.tcp.org/hspec.html/`, 2017.

[16] Witten, I. H.—Neal, R. M.—Cleary, J. G.: Arithmetic Coding for Data Compression. Communications of the ACM, Vol. 30, 1987, No. 6, pp. 520–540, doi: 10.1145/214762.214771.

[17] Ziv, J.—Lempel, A.: A Universal Algorithm for Sequential Data Compression. IEEE Transactions on Information Theory, Vol. 23, 1977, No. 3, pp. 337–343.

[18] Ziv, J.—Lempel, A.: Compression of Individual Sequences via Variable-Rate Coding. IEEE Transactions on Information Theory, Vol. 24, 1978, No. 5, pp. 530–536.

[19] Zukowski, M.—Héman, S.—Nes, N.—Boncz, P.: Super-Scalar RAM-CPU Cache Compression. Proceedings of the 22nd International Conference on Data Engineering (ICDE '06), IEEE Computer Society, 2006, doi: 10.1109/ICDE.2006.150.



Vinicius Fulber Garcia received his Bachelor's degree from the Universidade Federal de Santa Maria (UFSM), Rio Grande do Sul, Santa Maria, Brazil. Currently he is a Master's degree student at UFSM. His research interests include low-level optimization techniques, data compression, network security and network virtualization.

Sergio Luis Sardi Mergen received his Ph.D. degree in computer science from the Database Research Group at the Universidade Federal do Rio Grande do Sul (UFRGS), Rio Grande do Sul, Porto Alegre, Brazil, in 2011. His thesis was on on-the-fly integration and querying of structured data sources available on the Web. He is currently Professor of computing science at the Universidade Federal de Santa Maria (UFSM), Rio Grande do Sul, Santa Maria, Brazil. His research interests include algorithms, data integration and data compression.