The VLDB Journal (1997) 6: 26–39

The VLDB Journal
© Springer-Verlag 1997

Dictionary-based order-preserving string compression*

Gennady Antoshenkov

Oracle Corporation, New England Development Center, 110 Spitbrook Road, Nashua, NH 03062, USA; e-mail: gantoshe@us.oracle.com

Edited by M.T. Ozsu. Received 1 February 1995 / Accepted 1 November 1995

Abstract. As no database exists without indexes, no index implementation exists without order-preserving key compression, in particular, without prefix and tail compression. However, despite the great potentials of making indexes smaller and faster, application of general compression methods to ordered data sets has advanced very little. This paper demonstrates that the fast dictionary-based methods can be applied to order-preserving compression almost with the same freedom as in the general case. The proposed new technology has the same speed and a compression rate only marginally lower than the traditional order-indifferent dictionary encoding. Procedures for encoding and generating the encode tables are described covering such order-related features as ordered data set restrictions, sensitivity and insensitivity to a character position, and one-symbol encoding of each frequent trailing character sequence. The experimental results presented demonstrate five-folded compression on real-life data sets and twelve-folded compression on Wisconsin benchmark text fields.

Key words: Indexing – Order-preserving key compression

1 Introduction

Order-preserving string compression targets speed improvements and space conservation of the most intensely used databased components: indexes and sort. Like any compression, order-preserving compression saves space and data moves, but in addition, it enables the correct ordinal comparison of the compressed data items, increases the comparison speed, and thus extends the compression application to indexing, sorting, merging, aggregation, etc. (See the outline of compression application to query processing in Graefe 1993, Sect. 12.2.)

* Note by Jim Gray, Editor in Chief: Gennady Antoshenkov passed away after this article had been accepted for publication. Tamer Ozsu revised the final article to satisfy the referee's suggestions. I am sure Gennady would want us to thank Tamer for his extraordinary contribution to this paper

When applied to sorted indexes, order-preserving compression increases B-tree nodes' fanouts at all levels, yielding flatter B-trees, and hence gives better performance for a single key retrieval. Simultaneously, it reduces the overall index size, improving performance of range or full index scan. Even the simple forms of string prefix compression proved to be beneficial in reducing index sizes (Bayer and Unterauer 1977) and in speeding up quick sort algorithms (Bayer and Lin 1989). The other order-specific task of compressing composite keys was stated and resolved (Blasgen et al. 1977) by inserting some control characters into the composite string at regular intervals. The above methods, however, do not capture frequent patterns within strings and hardly exhaust even half of the compression potentials offered by more advanced techniques.

Among general string compression methods, Huffman encoding (Huffman 1952) delivers the optimal translation of a finite set of symbols into a target set of strings holding the prefix property (no string in the set is a prefix of any other string in this set). Huffman encoding is not order-preserving except for the case of binary target strings covered by the Hu-Tucker algorithm (Hu and Tucker 1971). A truly optimal translation of a sequence of symbols into a binary string representing some real interval contained in [0,1) is known as arithmetic encoding (see description and the original references of Bell et al. (1990), p. 108. Arithmetic encoding approaches entropy and is inherently order-preserving unless the order is sacrificed for some speed improvement.

Both Hu-Tucker and arithmetic encoding translate one symbol at a time and, because of this, are slow, especially arithmetic encoding. Moffat and Zobel (1992) have reported arithmetic compression being a factor of 40 slower than their dictionary approach. Faster compression is achieved by detecting frequent substrings of a source string and translating such groups of characters into a corresponding symbol – this is known as dictionary compression. Dictionary methods, including the most popular Lempel-Ziv compression (Ziv and Lempel 1978) are usually not capable of preserving the order. One exception is the case when a set of frequent source substrings holds the prefix property and also is a partitioning of a set of all source substrings. Another exception is an extension of the first one with common (zilch) symbols in order to combine a number of less frequent substrings into a group and hence balance the frequencies (Zandi et al. 1993).

To see the extra difficulty involved in order-preserving compression, consider encoding the words *table*, *their*, and *train*, assuming that the patterns *ta*, *the*, and *tr* frequently occur in some text source. Suppose we assign the symbols *1* for *ta*, *2* for *t*, *3* for *the*, and *5* for *tr*. Then the translations table → *1*ble, text→*2*ext, their→*3*ir, tiger→*2*iger, train→*5*ain compress multi-letter patterns into single symbols, but, unfortunately, put *2*iger in front of *3*ir and thus scramble the order. Assigning *4* for *t* does not help because it reverses *4*iger and *3*ir. Similarly, we compromise the order if we assign any single distinct symbol *X* for *th*: that→*X*at, their→*3*ir, this→*X*is. Observe that the cause of order scrambling is the assignment of different symbols to pattern strings extending each other: in our case patterns, *t*, *th*, and *the* cause order ambiguity between words starting with *ta*, ..., *tg* and *ti*, ..., *tz* as well as between *tha*, ..., *thd* and *thf*, ..., *thz*.

A simple way to avoid such ambiguity is to include the above extra patterns into a set of patterns to be encoded, i.e., to assign a sequence of symbols to the pattern sequence *ta*, ..., *tg*, *tha*, ..., *thd*, *the*, *thf*, ..., *thz*, *ti*, ..., *tz*. This string set holds the prefix property, provides order-preserving encoding, but does little for compression because 48 patterns other than *ta*, *the*, and *tr* occur infrequently and occupy 94% of symbol space. Zandi et al. (1993) improved this method by assigning special (zilch) symbols to contiguous groups of infrequent patterns: *tb*, ..., *tg*→Z_1, *tha*, ..., *thd*→Z_2, *thf*, ..., *thz*→Z_3, *ti*, ..., *tq*→Z_4, *ts*, ..., *tz*→Z_5 and encoding *t* with Z_1 if *t* is followed by *b*, ..., *g*, and *th* with Z_2 if *th* is followed by *a*, ..., *d*, etc. However, they still miss an opportunity to assign a single symbol G_1 for a continuous gap between *ta* and *thd*, etc., and reduce space for symbols needed only to support order preservation.

In this paper, we state and resolve a general problem of order-preserving dictionary compression, covering zilch encoding, string prefix compression, and composite key compression as particular cases and improving each of them. This way, a full power of the order-indifferent dictionary compression becomes applicable to the order-preserving case, including a high speed. We start with the premise that each source substring to be encoded with an individual symbol must be a prefix of an interval of some ordered partitioning of the source substrings into a finite number of intervals. The ordered mappings of symbols into each possible partitioning intervals set, and then further into a corresponding interval prefix, exhaust all order-preserving dictionary mappings.

We then define a class of compression-suitable encodings that do not suffer some obvious compression suboptimalities and can be easily constructed from the set of frequent substrings. For a given substring frequency distribution, we show how the balanced near-optimal encoding partitions can be produced and how a supplementary Hu-Tucker or arithmetic encoding can improve cases where dictionary encoding alone does not provide the desired balancing. It happens that the order-preserving dictionary mapping can also be applied in place of Hu-Tucker or arithmetic encoding for translation of symbols into a target string set (Antoshenkov et al. 1994), potentially exceeding Hu-Tucker's compression rate. At the end, we present the experimental results obtained for real-life data sets and for Wisconsin benchmark (Gray 1993) using our prototype implementation.

Section 2 of this paper specifies the terms and scope of order-preserving encoding. Sections 3–5 present the encode and decode procedures for position-dependent, position-independent and flexible encodings, respectively. Section 6 explores structural properties of compression-suitable encode vectors. Section 7 describes algorithms for optimal dictionary generation. In Sect. 8, blending techniques and their impact on compression are discussed. Section 9 presents experimental results, and Sect. 10 concludes the paper.

2 String encoding

Alphabet A is a non-empty ordered set of characters $A = \{a_1, \ldots, a_N\}$. A string is an ordered chain "$a_{i_1} a_{i_2} \ldots a_{i_l}$" of characters a_{i_j} drawn from some alphabet. $len(s)$ is the length of string s, for an empty string $len(\text{" "}) = 0$. $substr(s, i, l)$ is a substring of string s starting from position $i \geq 1$ and stretching up to l characters to the right, $substr(s, i) = substr(s, i, len(s))$. String v is a prefix of string w if $substr(w, l, len(v)) = v$. For a non-empty string set S, a common prefix of S, $compref(S)$, is the longest string being a prefix of all strings in S. String w extends string v if v is a prefix of w and $v \neq w$.

Examples. Strings " ", "*t*", "*th*", "*the*" are prefixes of "*the*". String "*train*" extends " ", ..., "*trai*", but not "*train*". String "*th*" is a common prefix of "*that*" and "*this*", *compref({"their", "tiger", "tee"}) = "t". substr("table", 2) = substr("table", 2,5)="able".*

String comparison is defined to be equal for identical strings, greater for a string extending another string, and otherwise according to comparison of the next character after a common prefix. To facilitate an SQL-style, "pad-character" comparison, we assume that all strings to be compared are extended to the right with a pad character to the common length L.

Having a common string length in order-related data manipulation and structures is important for sorting and index creation when keys are composed of multiple fields. Multi-field keys in such cases can be simply defined as a string concatenation of individual fields and be compressed, processed, and stored as a single string. Later, it will become obvious that the concatenated strings can have the pad character sequences of individual fields compressed into a single code for arbitrary common length L, making pad-character string manipulation as efficient as processing variable-length strings.

Let $S^L = \{\text{"}a_{i_1} \ldots a_{i_L}\text{"}\}$ be a non-empty subset of all strings with length L, a_{i_j} drawn from alphabet $A = \{a_1, \ldots, a_N\}$. Let $T = \{\text{"}b_{i_1} b_{i_2} \ldots\text{"}\}$ be a set of all strings on alphabet $B = \{b <_1, \ldots, b_M\}$. Encoding the source set S^L by the target strings from T is a transformation $t = encode(s)$ providing a one-to-one mapping of S^L into T. Decoding is done by a reverse transformation $s = decode(t)$. We are interested in the *order-preserving encoding* where for any $s_1, s_2 \in S^L$, $s_1 < s_2$ yields $encode(s_1) < encode(s_2)$.

We will explore a particular way of encoding where:

1. the source string is subdivided into several non-empty substrings taken from a given string vector Q = ($q_1, \ldots,$ q_K),
2. each substring is replaced with a corresponding symbol: position i of q_i in vector Q,
3. a sequence of symbols is translated into the target string set T using Hu-Tucker, arithmetic, or other encoding methods,
4. source-to-symbol and symbol-to-target translations are both order-preserving.

Suppose a source-to-symbol translation is defined that subdivides any non-empty source string into substrings q_{i_1}, …, q_{i_n} and produces the symbol chain $i_1, \ldots, i_n$. Since the first translated symbol i_1 takes its value from the set of integers $1, \ldots, K$, the defined above translation also defines a partitioning of all source strings into K classes, one per each value of the first translated symbol. Due to the order-preserving translation property, this partitioning must be a partitioning of source strings into K intervals following each other in the string order. To avoid ambiguity of the reverse mapping of the first symbol to the first substring (which would make decoding impossible), the first substring q_{i_1} must be a prefix of a common prefix of the interval corresponding to i_1. When the source string subdivision involves more than one substring, the described above partitioning and prefix rules must also be applied to the remaining translation of $q_{i_2}, \ldots, q_{i_n}$ into $i_2, \ldots, i_n$, and so on.

Partitioning into intervals and selection of interval prefixes at each step of translation might depend on the character position of the current substring in the source string and also on the values of already processed substrings. Further, we will consider only the character position dependency and independency cases. With this very general definition of order-preserving dictionary encoding, the relationship between partition intervals and their common prefixes is fairly intricate and is a topic of our compression optimality investigation.

3 Position-dependent encoding

In this and the following two sections we introduce several order-preserving encoding and decoding mechanisms. These mechanisms are driven by sets of encode/decode vectors which can be chosen with a great degree of freedom and can compress or expand source data depending on vectors' selection. How to select vectors delivering good compression is discussed in Sects. 6–8.

For a source string set S^L, we define trailing substring sets $S^L_j = \cup_{s \in S^L} substr(s, j)$, $1 \leq j \leq L$. Like S^L, each S^L_j is a non-empty set of equal-length strings, $S^L_1 = S^L$. Let S be one of equal-length trailing substring sets. Let $E = ([l_1, r_1], \ldots, [l_K, r_K])$ be an *encode vector*: an ordered partitioning of S into K closed intervals, $l_1 \leq r_1 < \ldots < l_K \leq r_K$, $l_i, r_i \in S$, $compref(l_i, r_i) \neq$ " " for $l \leq i \leq K$. Let $Q = (q_1, \ldots, q_K)$ where q_i is a non-empty prefix of $compref(l_i, r_i)$ be an *encode prefix vector*. Let $D = (d_1, \ldots, d_K)$ be a *decode vector* where d_i contains all information needed for encoding symbol i into a target string t using a given symbol-to-target encode method. For Hu-Tucker encoding, $(d_i, \ldots, d_K)$ is the ordered set of strings from the target binary alphabet that holds the prefix property. For arithmetic encoding, $(d_1, \ldots, d_K)$ is the ordered set of intervals d_i that constitute a partitioning of the interval $[0, 1)$.

Given a set $C = \{(E_1, Q_1, D_1), \ldots, (E_L, Q_L, D_L)\}$ of encode/prefix/decode vectors with equal vector length in each triplet, one triplet for each equal-length trailing substring set, the *encode* transformation of S^L into T is defined as follows:

```
Procedure $t$ = $encode(s)$

1. Take $s \in S^L$ as a source string, set $j$ = 1, $t$ = " ".
2. In $E_j$, find $[l_i, r_i]$ to which $substr(s, j)$ belongs.
3. Extend $t$ using $d_i$ from vector $D_j$.
4. Increment $j$ by $len(q_i)$.
5. If $j \leq L$, switch to step 2.
6. Return $t$ as the encoded string.
```

This encode procedure performs L or fewer iterations because all common prefixes in step 4 are non-empty strings having greater than zero length. A sequence G of j values with one value picked at the start of each iteration (step 2), defines a subdivision of s into the "source code" substrings.

Example. We want to encode positive 8-digit binary numbers, assuming that numbers with a small amount of significant digits occur more often than those with many significant digits.

Using the encoding scheme in Table 1, the following numbers are decomposed and encoded like:

$$\text{00000001}\| \rightarrow \text{0}\| \ , \quad \text{000001}\|\text{0}\|\text{1}\| \rightarrow \text{101}\|\text{0}\|\text{1}\| \ ,$$
$$\text{01}\|\text{1}\|\text{1}\|\text{0}\|\text{0}\|\text{0}\|\text{1}\| \rightarrow \text{11010}\|\text{1}\|\text{1}\|\text{0}\|\text{0}\|\text{0}\|\text{1}\| \ .$$

This scheme emulates δ encoding (Elias 1975), widely used for compressing sets of integers and for run-length bitmap compression. Our encoding simply flips some leading bits in the leading prefix compared to δ encoding and thus achieves order preservation, retaining the same compression rate as in δ.

Thereom 1. *The encode transformation defined above is order-preserving.*

Proof. Let s_1, s_2 be different strings from S^L, $s_1 < s_2$, and j_d be the biggest value in the sequence G subdividing string s_1 which satisfies $j_d \leq len(compref(s_1, s_2) + 1)$ condition. At the start of the iteration having $j = j_d$, *encode* will have the identical source codes processed for s_1 and s_2 and thus have identical partial strings t built in both cases. At $j = j_d$ iteration, the source codes for s_1 and s_2 are different. Moreover, the s_1 source code precedes the s_2 source code in the string order because the first differentiating character of s_1 and s_2 at position $len(compref(s_1, s_2) + 1)$ belongs to both these codes. It follows that symbol i for a differentiating source code of s_1 is smaller than that of s_2 and that partial strings t built at j_d iteration preserve the order. The rest of the translation does not change this order preservation, making the theorem proof complete.

Table 1. In this encoding scheme, the leading edge encode vector transforms leading zeros followed by the most significant '*1*' into the target codes of the decode vector D_1. Other trailing substring encodings are defined as an "identity" transformation. Longer source prefixes produce shorter codes

	q_i = $compref(l_i, r_i)$		$[l_i, r_i]$	i		d_i
Q_1	0000001	E_1	[00000001,00000001]	1	D_1	0
	0000001		[00000010,00000011]	2		100
	000001		[00000100,00000111]	3		101
	00001		[00001000,00001111]	4		11000
	0001		[00010000,00011111]	5		11001
	001		[00100000,00111111]	6		11010
	01		[01000000,01111111]	7		1101
	1		[10000000,11111111]	8		1110000
Q_2	0	E_2	[0000000,0111111]	1	D_2	0
	1		[1000000,1111111]	2		1
Q_8	0	E_8	[0,0]	1	D_8	0
	1		[1,1]	2		1

Now, let t be the target string that encodes a source string s using the above set C of the encode/prefix/decode vectors. The *decode* transformation of the set $\overline{T}$ = $encode(S^L)$ of all such t's back into the source string set S^L is described as follows:

Procedure s = $decode(t)$

1. Take an encoded string $t \in \overline{T}$, set j = 1, s = " ".
2. In D_j, find d_i used for production of t.
3. Append q_i from Q_j to the end of s.
4. Reduce t according to d_i.
5. Increment j by $len(q_i)$.
6. If $j \leq L$, switch to step 2.
7. Return s as the decoded source string.

In the *decode* procedure, decomposition of t = $encode(s)$ is done by iterating through the same sequence of source/target code pairs as during encoding of s. Indeed, the original symbol i is reconstructed in step 2 for each iteration because of reversibility of the symbol-to-target translation. Therefore, a corresponding source code interval $[l_i, r_i]$ must be the one used in the corresponding iteration of $encode(s)$, making the $decode$ loop a stepwise process of the original string s restoration.

4 Position-independent encoding

In the real world, ordered and to-be-ordered string sets have two types of frequently occurring patterns suitable for compression: (1) patterns specific to the character position in a string, and (2) patterns related to individual characters or correlated character sequences regardless of their position. Encoding of the second type needs a single encode/prefix/decode vector triplet to be applied at any source string position, and further, it needs a (single) set representing all trailing substrings for the encode vector definition.

Let S^L_* = $\cup_{1 \leq j \leq L} S^L_j$ be a set of all trailing substrings in S^L. Let S^L be a set of all $s \in S^L_*$ that are not extended by any other $s' \in S^L_*$. $\hat{S}^L$ holds the prefix property. Now we define the *encode vector* on $\hat{S}^L$ exactly like we did for S^L_j, i.e., as the ordered partitioning E = $([l_1, r_1], \ldots, [l_K, r_K])$ of $\hat{S}^L$ into trailing string intervals, with the left/right interval ends belonging to $\hat{S}^L$ and a common prefix for each interval being at least one character long. Similarly, we define a single Q as a set of non-empty prefixes of common prefixes of intervals in E and a single decode vector D.

Note that $\hat{S}^L$ is a maximal subset of S^L_* satisfying the prefix property. Partitionings E' defined similarly for any other subset of S^L_* with prefix property have to be part of a set of all maximal partitionings E and, therefore, $\hat{S}^L$ covers the most general case. Also, here and in other encodings, we choose for partitioning only the intervals whose ends belong to fully ordered trailing substring sets ($\hat{S}^L$ in this case). If we lift this restriction and allow the partitioning interval ends to take arbitrary values preserving the partitioning property, we will not add any new $\hat{S}^L$ partitionings to the restricted case, but merely extend the ways of representing the same set of all partitions E. This particular way that we chose is characterized by the equality $compref([l_i, r_i])$ = $compref(l_i, r_i)$, simplifying explanations and implementation as well.

Also, using such unique "end-inclusive" partition representation, a number of all possible partitions of $\hat{S}^L$ containing n strings can be easily established as 2^{n-1}. Indeed, the leftmost partition interval must have its left end coinciding with the leftmost string in $\hat{S}^L$. Any of the remaining $n - 1$ strings in $\hat{S}^L$ might or might not be the left end of some partitioning interval, independent of other $\hat{S}^L$ strings. Since the right partitioning interval ends can be always unambiguously determined given the set of all left interval ends, the desirable degree of freedom can be expressed as the number of $n - 1$ independent binary trials, i.e., as 2^{n-1}.

In order to apply the previously defined encode procedure to the position-independent case, we first introduce the notations that simplify string and interval comparisons. Given a string q, notation $lowpad(q)$ will stand for q padded to the right to length L with the lowest alphabet character and, similarly, $highpad(q)$ will denote highest character padding. Given a string interval $[l, r]$, the notation $\lfloor l, r \rfloor$ will stand for the closed interval $[lowpad(l), highpad(r)]$, and $\lfloor x \rfloor$ will stand for $\lfloor x, x \rfloor$. If s is a trailing substring in S^L_*, there is at least one string in $\hat{S}^L$ having s as a prefix, and, therefore, s is a prefix of some string q from some interval $[l_i, ri]$ in E. It follows that the intersection $[l_i, r_i] \cap \lfloor s \rfloor$ is not empty because q belongs to both intervals.

To define the *encode* procedure, we use $C = \{(E_1, Q_1, D_1)$, …, $(E_L, Q_L, D_L)\}$ with $E_1 = \ldots = E_L = E$, $Q_1 = \ldots = Q_L = Q$, $D_1 = \ldots = D_L = D$ of identical equal-length encode/prefix/decode vectors (of course, only one copy of each is used in implementation) and use the previously defined $encode(s)$ procedure with a modified step 2:

2. In E_j, find $[l_i, r_i]$ intersecting $\lfloor substr(s, j) \rfloor$.

If in step 2, the padded interval $\lfloor substr(s, j) \rfloor$ encloses $[l_i, r_i]$, then $substr(s, j)$ is a prefix of all strings in $[l_i, r_i]$ and thus is a prefix of $compref(l_i, r_i)$. In this case, if $q_i(compref(l_i, r_i)$, then the current iteration advances j by $len(q_i) = len(compref(l_i, r_i)) \geq len(substr(s, j)) = L - j + 1$, making j greater than L and, hence, leading to the procedure conclusion. Otherwise, if $q_i < compref(l_i, r_i)$, the procedure terminates according to Step 5. If, on the contrary, the above enclosure does not hold, then one end of interval $[l_i, r_i]$ lies outside $\lfloor substr(s, j) \rfloor$, i.e., there exists string $x : x \in \hat{S}^L$, $x \in [l_i, r_i]$, and $x \notin \lfloor substr(s, j) \rfloor$. Simultaneously, there exists string y common to both of them: $y \in \hat{S}^L$, $y \in [l_i, r_i]$, and $y \in \lfloor substr(s, j) \rfloor$. Indeed, if the other end of interval $[l_i, r_i]$ belongs to $\lfloor substr(s, j) \rfloor$, we pick it as y, otherwise $[l_i, r_i]$ encloses $\lfloor substr(s, j) \rfloor$ and we pick as y a string from $\hat{S}^L$ having $substr(s, j)$ as a prefix. This proves that $substr(s, j)$ extends $compref(l_i, r_i)$ and also extends q_i, requiring more iterations to complete encoding.

Example. The following position-independent encoding is defined for all 13-character-long strings on alphabet $A = (a, b, c)$. It assumes that (a) "b" stands for blank and is used as a single blank for word separation or as a trailing blank sequence, (b) the length of all "a" sequences is usually even, and (c) "a" sequences usually follow "c" sequences.

i		$q_i = compref(l_i, r_i)$	$[l_i, r_i]$
1	*	aa	$\lfloor aa \rfloor$
2		a	$\lfloor ab, a \rfloor$
3		b	$\lfloor b, bbbbbbbbbbbba \rfloor$
4	*	$bbbbbbbbbbbbb$	$\lfloor bbbbbbbbbbbbb \rfloor$
5		b	$\lfloor bbbbbbbbbbbbc, b \rfloor$
6	*	caa	$\lfloor caa \rfloor$
7		c	$\lfloor cab, cbcc \rfloor$
8	*	$ccaa$	$\lfloor ccaa \rfloor$
9		cc	$\lfloor ccab, cc \rfloor$

* marks frequent patterns. In these frequent patterns, consecutive as occur even number of times (twice) and, when seen together, as always follow cs

With this encode vector, string "*caaaabccaabbb*" is parsed and translated into a chain of symbols i as

caa	aa	b	$ccaa$	bbb	– tokens
6	1	5	8	4	– symbols

Here, at the first iteration, interval $\lfloor caa \rfloor$ is found to include $\lfloor caaaabccaabbb \rfloor$ delivering symbol 6. At the second iteration, $\lfloor aa \rfloor$ is found to include the remainder $\lfloor aabccaabbb \rfloor$ delivering 1. Further, $\lfloor bbbbbbbbbbbbc, b \rfloor$ is found to include $\lfloor bccaabbb \rfloor$ delivering 5. Note that in the order-indifferent compression, the inclusion $\lfloor b \rfloor \supset \lfloor bccaabbb \rfloor$ would select token "b" and eliminate the need for one of the two symbols 3 or 5 assigned to the specific intervals in our case. Further, $\lfloor ccaa \rfloor$ is found to include $\lfloor ccaabbb \rfloor$ delivering 8, and last, $\lfloor bbbbbbbbbbbbb \rfloor$ is found to be included in the remainder $\lfloor bbb \rfloor$. Also note that token "*bbbbbbbbbbbbb*" not only exhausts the source string remainder but also contains ten extra bs which are ignored by the encoding procedure because only a non-empty intersection of both intervals is required in the new step 2.

To define the $decode$ procedure for encode/prefix/decode vectors E, Q, D, we use the previously defined $decode(t)$ procedure with a modified step 7:

7. Return $substr(s, 1, L)$ as the decoded string.

Here a truncation of the final s to the standard length L is needed because the source code at the last iteration can be oversized (see the case of $\lfloor substr(s, j) \rfloor$ enclosing $[l_i, r_i]$ in step 2 of the $encode$ procedure).

5 Flexible encoding

Position-dependent encoding with different encode/prefix/ decode vectors required for each string position may not necessarily reflect the most practical case because the frequent patterns are usually bound to only a few particular positions. Take accounts, license plates, and other forms of identification in which strings are controlled by templates. There are perhaps a few positions designated to separators or specific code letters and maybe a few areas filled with digits, or letters only, or a mixture of both, possibly enhanced with some commonly used printable characters. What is needed to cover these cases is a flexible mixture of several position-dependent and position-independent encoding schemes.

For each string position j, $1 \leq j \leq L$, let J_j be a subset of integers between 1 and L containing j. We want to define such encoding that the source codes picked at position j are taken from the encode prefix vector based on trailing substrings starting at any position k in J_j, not just at position j. For this, we first redefine S^L_j to be a set of J_j-based trailing substrings: $S^L_j = \cup_{k \in J_j} (\cup_{s \in S^L} substr(s, k))$. Then, as in position-independent encoding, we define $\hat{S}^L_h$ as the largest subset of S^L_j with none of its strings extending the other. We then redefine E_j as the ordered partitioning of $\hat{S}^L_j$ into intervals in which the ends belong to $\hat{S}^L_j$ and in which common prefixes are at least one character long.

One can observe that the encode and decode procedures described for position-independent encoding, also encode/decode strings from S^L based on the code set $C = \{(E_1, Q_1, D_1), \ldots, (E_L, Q_L, D_L)\}$ where E_js are the J_j-based redefinition of encode vectors, and encode, prefix, and decode vectors are identical for every integer from a given J_j. When each J_j contains only one integer, i.e., $J_j = \{j\}$, this "flexible" encoding becomes a position-dependent encoding. When $J_1 = \ldots = J_L = \{1, 2, \ldots, L\}$, $E_1 = \ldots = E_L$, $Q_1 = \ldots = Q_L$, and $D_1 = \ldots D_L$, the flexible encoding becomes a position-independent encoding.

Further we will investigate properties of flexible encoding useful for the task of compression. The results will be equally applicable to the extreme cases of position-dependent and position-independent encoding.

6 Source code selection

In this and the following two sections we will concentrate on properties and algorithms related to optimality of the source side (i.e., a dictionary portion of) compression. Our first observation is that the cases where prefixes q_i are shorter than $compref(l_i, r_i)$ deliver compression typically inferior to those cases, where $q_i = compref(l_i, r_i)$. Indeed, except for some rare parsing patterns, substitution of longer sequences with the same number of symbols as for shorter sequences improves the compression factor. A possibility of encoding prefixes shorter than $compref(l_i, r_i)$ might turn out to be useful in some future applications, but maximization of compression rate calls for discarding these cases.

Our second observation is that if two different encoding schemes break all source strings into identical sequences, a scheme with fewer intervals in its encoding vector gives better compression. Let $[l_i, r_i]$ and $[l_{i+1}, r_{i+1}]$ be two adjacent intervals of an encode vector E, x and y be common prefixes of these intervals, and $z = compref(x, y)$. According to the common prefix definition, each of x, y either extends or is equal to z. Suppose $x = y = z$. When either of the two adjacent intervals is selected in step 2 of the $encode(s)$ procedure, the same source code z is used for encoding. If E' is the encode vector derived from E by substituting two intervals $[l_i, r_i]$, $[l_{i+1}, r_{i+1}]$ with one interval $[l_i, r_{i+1}]$, a selection of $[l_i, r_i]$ or $[l_{i+1}, r_{i+1}]$ from E yields the same code z picked as in the selection of $[l_i, r_{i+1}]$ from E'. However, vector E is one interval longer than E', causing inferiority of E-based compression.

Now we can define *compression-suitable encoding* as encoding by $C = \{(E_1, D_1), \ldots, (E_L, D_L)\}$, where (a) prefix vectors $Q_1, \ldots, Q_L$ are implicitly derived from $E_1, \ldots, E_L$ by setting all q_is to corresponding $compref(l_i, r_i)$ and (b) at least one of the two intervals in each adjacent interval pair in all E_js has its common prefix extending the common prefix of the interval pair.

There are three encode vector interval types in compression-suitable encoding distinguishable in the context of left and right neighboring intervals. If intervals I_x, I_y, I_z are adjacent and have common prefixes x, y, z, interval I_y is called:

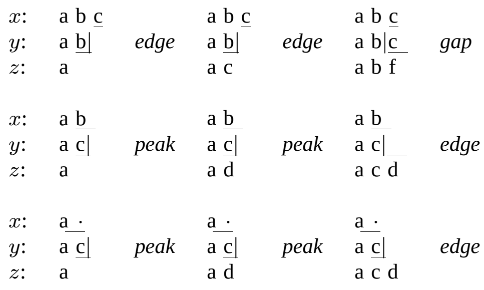

Fig. 1. Classification of extension patterns into *peak/edge/gap* types

peak if y extends $compref(y, x)$
and y extends $compref(y, z)$,

edge if x extends $compref(x, y)$
and y extends $compref(y, z)$
or
z extends $compref(z, y)$
and y extends $compref(y, x)$,

gap if x extends $compref(x, y)$
and z extends $compref(z, y)$.

Interval types for all extension patterns of x, y, z are depicted in Fig. 1.

For the first (or last) interval in E we will use the same classification assuming a phantom interval I_x (or I_z) with empty string " " as its common prefix. The end intervals cannot be gaps.

Note that the common prefix y of edges and gaps is extended by x or z and thus is not part of the underlying set $\hat{S}_j^L$ because of its prefix property. For gaps it means that interval I_y must contain at least two strings s_1, s_2 extending y such that $y = compref(s_1, s_2)$, hence the characters a_1, a_2 immediately following prefix y in s_1 and s_2 must be different. For example, the gap common prefix "*ab*" in Fig. 1 must contain in its interval a string starting with "*abd*" and a string starting with "*abe*". In the case of binary alphabets, the encode vectors in compression-suitable encoding contain only peaks.

Theorem 2. *If E is an encode vector of a compression-suitable encoding, then the following properties hold.*

A. *A set of common prefixes of all peaks in E is the largest set of common prefixes of E intervals holding the prefix property.*

B. *For every pair of neighboring peaks I_x, I_y (no other peaks lie in between) with common prefixes x, y, a sequence of intermediate intervals is composed of*

  1. *possible sequence of (right) edges $I_{r_1}, \ldots, I_{r_m}$ with common prefixes $r_1, \ldots, r_m$, x extending r_1 extending $\ldots$ extending r_m extending $compref(x, y)$, followed by*

  2. *possible gap I_g with its common prefix $g = compref$ (x, y), followed by*

  3. *possible sequence of (left) edges $I_{l_1}, \ldots, I_{l_n}$ with common prefixes $l_1, \ldots, l_n$ $compref$ (x, y) extended by l_1 extended by $\ldots$ extended by l_n extended by y.*

C. *The first (last) peak is preceded with (followed by) a possible sequence of left (right) edges like in B(3) (B(1)).*

Proof. Let y be a common prefix of some peak interval I_y in E, x and z be common prefixes of the peak's left and right adjacent intervals I_x, I_z. Notice that because y extends $compref(y, x)$, y also extends $compref(y, s)$ for common prefix s of all intervals in E to the left of I_x and similarly for intervals to the right of I_y. Notice further that if y extends $compref(y, s)$ for string s, then s does not extend y because if it did, $y = compref(y, s)$ would hold, which is incorrect. From the above, it follows that y is not extended by the common prefix of any interval in E. This proves a set of peaks in E to hold the prefix property. The largest set of E-interval common prefixes, which holds the prefix property, is obviously a set of all such common prefixes that are not extended by any E-interval common prefix. Non-peak intervals have their prefixes extended by either left, right, or both adjacent interval common prefixes and, therefore, do not belong to the above set. This completes the proof of property **A**. From the property **A**, it follows that common prefixes of all intervals between any two neighboring peaks I_x, I_y do not belong to the largest set holding the prefix property, and, therefore, each intermediate prefix is extended by some peak, and further, either I_x, or I_y, or both are the extending peaks. Sequencing of intermediate edges and gaps described in **B1, 2, 3** is a straightforward derivation from the peak/edge/gap definition. The initial and concluding cases of property **C** sequencing are proved similarly.

Having described the sequencing patterns of compression-suitable encode vector, we now want to explore the reverse task of determining what string vector can serve as a vector of common prefixes of some compression-suitable encode vector.

Let $\hat{S}$ be a set of strings on alphabet A holding the prefix property, x, y, z be a sequence of A strings, $y \neq$ " ". By convention, $(x, y,$ " "$)$, $($" "$, y, z))$, $($" "$, y,$ " "$)$ will represent shorter sequences (x, y), (y, z), (y). For y in sequence (x, y, z) we define *string-extending interval narrowed by adjacent extension* $SEINAE(x, y, z)$ or simply $SEINAE(y)$ for an implicit adjacent string context as follows. If

1. there exists the leftmost string l in $\hat{S}$ such that $l \geq lowpad(y)$ and, when x is not extended by y, $l > highpad(x)$, and
2. there exists the rightmost string r in $\hat{S}$ such that $r \leq highpad(y)$ and, when z is not extended by y, $r < lowpad(z)$, and
3. $l \leq r$,
   then $SEINAE(y) = [l, r]$, otherwise $SEINAE(y)$ is said to be nonexistent.

When it exists, $SEINAE(y)$ is contained in $\lfloor y \rfloor$ – this follows from $l \geq lowpad(y)$ and $r \leq highpad(y)$. Also, $SEINAE($" "$, y,$ " "$)$ is the largest interval $[l, r]$ contained in $\lfloor y \rfloor$ with $l, r \in \hat{S}$. In addition, $SEINAE(x, y, z)$ can only be smaller (more narrow) than $SEINAE($" "$, y,$ " "$)$ if one of x, z extends y.

Theorem 3. *Let $\hat{S}^L_j$ be the largest subset of all J_j-based trailing substrings of S^L holding the prefix property as defined in flexible encoding and $Q_j = (q_1, \ldots, q_K)$ be a vector of non-empty strings on the same alphabet. If for each q_i in Q_j there exists $SEINAE(q_i)$ such that $q_i = compref(SEINAE(q_i))$ and $\hat{S}^L_j \subset \cup_{1 \leq i \leq K} SEINAE(q_i)$, then $E_j = (SEINAE(q_1), \ldots, SEINAE(q_K))$ is the compression-suitable encode vector and Q_j is a vector of common prefixes of E_j's intervals.*

Proof. First, we verify that if an arbitrary string x_i is picked from each $SEINAE(q_i)$ and $x_i \in \hat{S}^L_j$, then $x_1 < \ldots < x_K$ holds. Suppose that the ascending order does not hold for $x_1, \ldots, x_K$. Then there must be an adjacent string pair x_i, x_{i+1} such that $x_i \geq x_{i+1}$. If q_i is not extended by q_{i+1}, then all strings in $SEINAE(q_{i+1})$ follow $highpad(q_i)$, which contradicts $x_i \geq x_{i+1}$. Otherwise, q_{i+1} is not extended by q_i and all strings in $SEINAE(q_i)$ precede $lowpad(q_{i+1})$, which again contradicts $x_i \geq x_{i+1}$. This proves that intervals $SEINAE(q_i)$ do not intersect each other and are ordered in the ascending string sequence.

Since the union of $SEINAE(q_i)$ contains $\hat{S}^L_j$, vector $E_j = (SEINAE(q_1), \ldots, SEINAE(q_K))$ defines the ordered partitioning of $\hat{S}^L_j$. Further, from $q_i = compref$ $(SEINAE(q_i))$ it follows that Q_j is the vector of common prefixes of E_j intervals. Finally, since Q_j contains only non-empty strings, E_j is the encode vector.

To verify that E_j is compression-suitable, let us assume the opposite, that some adjacent interval pair $SEINAE(q_i)$, $SEINAE(q_{i+1})$ has both its common prefixes $q_i = q_{i+1} = compref(q_i, q_{i+1})$. In accordance with $SEINAE$ definition and given that q_i is not extended by q_{i+1}, all strings in $SEINAE(q_{i+1})$ should follow $highpad(q_i) = highpad(q_{i+1})$, which is impossible. This completes the theorem proof.

With the results stated in Theorem 3, it becomes possible to use a string vector Q_j as a shorthand for any given compression-suitable encode vector E_j. Even more importantly, one can easily pick a set of the most frequent substrings, add other substrings to fill the gaps between them producing Q_j, and reconstruct the encode vector E_j from Q_j by applying $SEINAE(q_i)$. This methodology is at the heart of any manual construction of an encoding scheme (example follows) and it is also heavily used in the algorithms that automatically produce vectors E_j and Q_j, like those presented in the next section.

Example. Suppose we want to compress a set of all four-letter strings on alphabet $A = (a, b, c, d)$ starting with "*a*", knowing that substrings "*aba*", "*abd*", "*ac*", "*adbb*", and substring "*adb*" followed by letters "*c*" and "*d*" occur often in our source string set. Let us examine the neighborhoods of these frequent strings going in the ascending string order and constructing prefix and encode vectors.

First, we discover that there is a gap in front of "*aba*" containing the lowest string "*aaaa*". According to Theorem 3, "*aaaa*" must be part of the union $\cup_{1 \leq i \leq K} SEINAE(q_i)$ of all selected prefixes, but it is not. Therefore, our set of frequent prefixes must be extended with either of the infrequent prefixes "*a*" or "*aa*". $SEINAE($" "$,$ "*aa*"$,$ "*aba*"$)$ = ["*aaaa*", "*aadd*"] with $compref($"*aaaa*"$,$ "*aadd*"$)$ = "*aa*", because "*aa*" is not narrowed by its right neighbor "*aba*", neither is it affected by " " since "*aa*" extends " ". This makes "*aa*" the right choice for constructing a good prefix set. Note that by filling the leftmost gap we have, in fact, created a peak.

Table 2. * marks the most frequent (initial) prefixes. Other (infrequent) prefixes merely fill the gaps

	Q_1	E_1	i
peak	$\mathrm{a}\ \underline{\mathrm{a}}\vert\underline{\quad\cdot}$	[aaaa, aadd]	1
peak *	$\mathrm{a}\ \mathrm{b}\ \underline{\mathrm{a}}\vert$	[abaa, abad]	2
gap	$\mathrm{a}\ \mathrm{b}\ \lfloor\underline{\ \ \cdot}$	[abba, abcd]	3
peak *	$\mathrm{a}\ \underline{\mathrm{b}\ \mathrm{d}}\vert$	[abda, abdd]	4
peak *	$\mathrm{a}\ \underline{\mathrm{c}}\vert$	[acaa, acdd]	5
edge	$\mathrm{a}\ \mathrm{d}\vert\underline{\quad\cdot}$	[adaa, adba]	6
peak *	$\mathrm{a}\ \mathrm{d}\ \mathrm{b}\ \underline{\mathrm{b}}\vert$	[adbb, adbb]	7
edge *	$\mathrm{a}\ \mathrm{d}\ \underline{\mathrm{b}}\vert$	[adbc, adbd]	8
edge	$\mathrm{a}\ \mathrm{d}\vert$	[adca, addd]	9

Second, we discover a gap between "aba" and "abd". Their common prefix "ab" immediately gives us a correct fill because $SEINAE$("aba", "ab", "abd") = ["$abba$", "$abcd$"] with $compref$("$abba$", "$abcd$") = "ab". This time we have constructed a real gap narrowed from both sides by the neighboring prefixes.

The next pair of frequent prefixes "abd" and "ac" has no gap between the adjacent ends of their extended intervals $\lfloor$"abd"$\rfloor$ and $\lfloor$"ac"$\rfloor$ and thus does not need a filling prefix. On the contrary, the next pair "ac" and "$adbb$" prefix has a gap with possible fills "a" and "ad". Similar to the first case, the longer prefix "ad", becomes our choice because $SEINAE$("ac", "ad", "$adbb$") = ["$adaa$", "$adba$"] and $compref$("$adaa$", "$adba$") = "ad". The last pair, "$adbb$" and "adb" is different from all the previous cases because the first string extends the second. Absence of a gap between them is due to this extension, not to the fact that they are adjacent like the pair "abd" and "ac". Finally, the gap after the last frequent prefix "adb" has two potential fillers, "a" and "ad". Our choice, as usual, is the longest prefix, "ad", because $SEINAE$("adb", "ad", " ") = ["$adca$", "$addd$"] and $compref$("$adca$", "$addd$") = "ad". Note that "ad" is an edge.

All frequent and newly added, infrequent prefixes are summarized in Table 2. Note that in this example, only one infrequent edge "ad" is produced to fill the gap between the two frequent peaks "ac" and "$adbb$". This is not possible with zilch encoding (Zandi et al. 1993), which requires two prefixes "ad" and "adb" to fill the gap between these frequent peaks. In general, zilch encoding is a particular case of peak/edge/gap prefix vector where none of the gaps or edges are extended with more than one character by a neighboring peak or edge.

A much more vivid difference between our compression and zilch compression can be observed on indexes containing many duplicate keys. If a 30-character country name is part of a large composite index, then each of about 200 country names can be encoded with a single symbol. Filling the gaps with our method would require another 201 infrequent gap prefixes yielding a total of 401 symbols, whereas zilch encoding would consume about 27 symbols on each side of each country name, yielding 11 000 symbols. If a symbol space is limited to 401 symbols, the zilch method will be able to encode only $\frac{1}{27}$ of all names and thus lose up to 27 times in compression rate. With no limit, however, zilch will consume on the order of 1 MB of main memory, removing this valuable resource from usage by other system components.

7 Toward optimal compression

Order-preserving dictionary encoding is essentially a repetitive process of identifying to which partitioning interval a given trailing substring belongs. It amounts to a series of character comparisons or character vector lookups that advance very quickly along the source string. If for a given trailing substring, there are two or more code prefixes in vector Q that extend each other and also match the trailing substring, the choice of a code prefix is determined unambiguously because we do not match prefixes with substring, but rather find the encode interval containing this substring and then pick the corresponding common prefix. The order-indifferent dictionary methods, however, always face this ambiguity problem and either ignore it by "greedy" selection of the longest matching prefix or optimize it by considering the alternatives and choosing the one yielding better compression for several encoding steps ahead. Our method needs no such parsing optimization and thus, in this respect, remains fast and optimal at the same time.

During parsing, once a partition is found, its symbol can be used for encoding as a target character, having the symbol chain serve as a translated target string, hence avoiding the symbol-to-target translation phase. Such dictionary-only encoding is very fast, but not necessarily the most compact method. We will discuss the tradeoffs between the dictionary-only and dictionary-and-target compression methods at the end of this section.

To measure the dictionary-only compression rate, we will use a frequency table containing pairs (s, f) of all or randomly-selected source substrings and their frequencies of appearance in an existing data set or the anticipated frequencies in a non-existing data set. The set of substrings in the frequency table can be assumed to hold the prefix property (see the technology of reassigning the shorter string frequencies in the next section). If N is the source alphabet size, K is the length of the encode vector $([l_1, r_1], \ldots, [l_K, r_K])$, f_i is the sum of frequencies of all table substrings belonging to $[l_i, r_i]$, and $q_i = comrpef(l_i, r_i)$, then the space consumption of the to-be-encoded prefixes q_i aggregated for all substrings in the frequency table is $\log_2 N \cdot \sum_1^K (len(q_i) \cdot f_i)$ bits, the corresponding space consumption of the encoded symbols is $\log_2 K \cdot \sum_1^K f_i$ bits, and the compression factor is

$$\frac{\log_2 K \cdot \sum_1^K f_i}{\log_2 N \cdot \sum_1^K (len(q_i) \cdot f_i)} .$$

Since for a given source alphabet and a given frequency table $\log_2 N$ and $\sum_1^K f_i$ are constants, the task of minimizing the compression rate is the task of maximizing $\sum_1^K (len(q_i) \cdot f_i) / \log_2 K$. It is also clear that only compression-suitable encode vectors should be considered. The following "greedy" algorithm builds an encode vector from the frequency table and is a close analogue of the traditional dictionary builder described by Bell et al. (1990,

p. 214). As in any dictionary method, a wide usage of a trie[1] structure for a string set traverse is assumed.

Algorithm 1(greedy)

1. Sort strings in the frequency table. Eliminate strings extended by other strings and distribute the eliminated string frequencies among the surviving strings (see blending methods below). Represent the frequency table as a sorted trie.
2. Define the initial encode vector as the partitioning intervals surrounding each source alphabet character.
3. Find the best local subdivision of each partitioning interval using Routine A.
4. Among all partitioning intervals, find the one in which the best subdivision delivers the best compression.
5. At some point, when a trend to a degrading compression rate is discovered or the encode vector length exceeds some limit, look at the partitioning history, pick the most satisfactory encode vector, and terminate the algorithm.
6. Materialize the globally best interval subdivision and find the fresh best local subdivitions of the two or three new-built intervals, using Routine A.
7. Go to step 4.

Routine A (interval subdivision)

For a given compression-suitable encode vector and for a given interval in this vector, consider each subdivision of this interval into either:

1. two intervals which, combined with the rest of the vector, become a new compression-suitable partitioning or,
2. three intervals which again yield a compression-suitable partitioning with the middle interval being a peak.

Among these subdivisions pick the one with the best compression (use frequency table for calculation) and store useful knowledge about this subdivision along with the given interval. (The above task can be performed with a single trie traverse.)

The mechanics of subdivision are fairly simple.

1. Suppose peak “the” is already part of the current partitioning and “$their$” is the most frequent extension of it. Then we replace a single peak “the” with the three prefixes: edge “the”, peak “$their$”, and edge “the”. The corresponding encode intervals will be: $SEINAE$(“ ”,“the”,“$their$”), $SEINAE$(“the”,“$their$”,“the”), and $SEINAE$(“$their$”, “the”, “”) or, if expressed directly, [“$theaaaa$”, “$theiqzz$”], [“$theiraa$”, “$theirzz$”] and [“$theisaa$”, “$thezzzz$”]. (Here we assume 7-character strings from the alphabet (“a”, $\ldots$, “z”).)

2. Suppose peak “$their$” is part of the current partitioning, gap “th” immediately follows “$their$”, and extensions of “the” to the right of “$their$” have high frequency. Then, subdivision of gap “th” caused by the introduction of “the” will include two prefixes: edge “the” and gap “th”. Similar subdivision will take place if “$their$” or “th” are the right edges.

3. Having peak “sin” followed by gap “si” followed by peak “sir” as part of the current partitioning and a new frequent prefix “sip”, we can subdivide “si” into two peaks, “sio” and “sip”, producing a continuous sequence of peaks: “sin”, “sio”, “sip”, and “sir”.

[1] Trie is a multiway tree with each path corresponding to a different string

In all these cases, we insert a new prefix into the current partitioning and check if extra prefixes have appeared around the new one; if they have appeared, we add them to the partitioning, calculate $SEINAE$ for all neighboring prefixes involved, and, finally, calculate $compref(SEINAE(\ldots))$ to extend some unnecessarily short prefixes. For other cases, which are not presented here, the procedure remains the same.

With each new subdivision, the new intervals need a fresh subdivision recalculation because new common prefixes q_i have new lengths and those are directly involved in compression rate calculation. However, intervals become progressively smaller and, as in a quick sort, require $O(K^* \log(K))$ average frequency table tests and $O(K^2)$ worse case tests. Unlike the traditional dictionary builders, the greedy algorithm considers long strings simultaneously with short strings, not all digrams followed by trigrams, etc. This makes a “greedy” advancement closer to the optimal gradient. It also avoids the traditional removal of the redundant short strings from the dictionary.

The near-gradient advancement manifests a desire to keep a partition weight $len(q_i) \cdot f_i$ balanced (i.e., about equal) between the partitioning intervals. A smaller-than-mean partition weight would potentially benefit from the interval merging and a higher-than-mean weight is a good target for splitting. In the greedy algorithm, the underweight/overweight avoidance is enforced by a local split toward the global gradient. The same balancing effect can also be achieved by a globally-equalized partitioning selection.

Algorithm 2 (equalizing)

1. Sort strings in the frequency table. Eliminate strings extended by other strings and distribute the eliminated string frequencies among the surviving strings (see blending methods below). Represent the frequency table as a sorted trie.
2. Pick some limit weight value W – a best guess for further steps to deliver a reasonably long encode vector.
3. Find the longest compression-suitable encode vector containing only peaks, gaps, and possibly one edge at each end, each peak having $len(q_i) \cdot f_i \geq W$. (This can be done with a single trie traverse.) For each peak, check the frequency table strings along the gaps or end edges adjacent to this peak going away from the peak, and find the first split that creates a neighboring edge on each peak side (if possible), having $len(q_i) \cdot f_i \geq W$. Continue checking and building such neighboring edges until exhaustion. At this point a compression-suitable encode vector is built with all peaks and edges, possibly excluding end edges, satisfying the $len(q_i) \cdot f_i \geq W$ condition.
4. Set new value W according to some strategy of exploring different encode vector lengths and compression factors with the goal of finding the most satisfactory encoding. If there is no or too little room for encoding improvement, pick the most satisfactory encode vector and terminate the algorithm.
5. Go to step 3.

An illustration of the workings of this algorithm can be found in the example summarized in Table 2.

Equalizing algorithm follows the peak/edge/gap structure of Theorem 2 and considers short prefixes after the longer ones – an ideal sequence to proceed. It is simpler to implement than the greedy algorithm because no priority queue for the globally best interval needs to be maintained. The overweight interval avoidance is embedded in the long-to-short prefix processing sequence and the underweight avoidance is rooted in enforcing the same limit W across all purposely created peaks and edges.

Both encode vector building algorithms tend to balance interval weights $len(q_i) \cdot f_i$. Recall for a moment that the space consumption of the to-be-encoded prefix q_i aggregated for all frequency table substrings is $\log_2 N \cdot \sum_1^K (len(q_i) \cdot f_i)$ bits. For substrings belonging to an individual interval $[l_i, r_i]$, this consumption amounts to $\log_2 N \cdot len(q_i) \cdot f_i$ bits or $len(q_i) \cdot f_i$ source characters. The probability that the character to be consumed by $([l_1, r_1], \ldots, [l_K, r_K])$ encoding belongs to the prefix q_i of interval $[l_i, r_i]$ is

$$p_i = \frac{len(q_i) \cdot f_i}{\sum_1^K (len(q_i) \cdot f_i)} .$$

Since the average space needed for one source character encoding is the entropy of $-\sum_1^K (p_i \cdot \log_2(p_i))$ bits and since the smallest possible entropy is reached with $p_1 = \ldots = p_K$, the most compact dictionary-only encoding by K intervals is achieved with the perfect weight balance $len(q_1) \cdot f_1 = \ldots = len(q_K) \cdot f_K$. This proves that by balancing interval weights the greedy and equalizing algorithms steer the encode vector search toward optimal compression. In several cases, however, a significant variance of weights is unavoidable:

1. Presence of source strings with a very high frequency creates peaks with non-restricted overweights.
2. Gap overweight, compared with the balance goal W, can be up to a factor of N (N is a source alphabet size).
3. Gap underweight can be unlimited, e.g., when a few or no substrings are present between two frequent “peak” strings.
4. There is also a noise-level variance of interval weights with a two-fold order of deviation.

Significant gap underweight in case (3) requires a long target code to balance the infrequency of such gap which, most of the time, is surrounded by frequent peaks or edges. It happens that the Hu-Tucker target encoding can hardly assign one long code (for the gap) between two neighboring short non-gap codes.[2] Arithmetic codes, on the other hand, can easily produce interchanging long/short code chains. For very infrequent gaps (and there are usually many of them), the best compression is achieved by assigning very long target codes to them. The pitfall here is that in such cases the encoded string may become very long and we would have to assign huge buffers to hold the maximum length encoded strings. Index retrieval software components usually have many such buffers, making it impractical to allow the worst case code expansion to be more than two-three times. Note that the traditional order-indifferent encodings either ignore the bad cases, because they are extremely rare, or designate one leading bit to indicate usage of encoded or original string, whichever is smaller. With the order-preserving encoding, we can neither ignore nor select the shortest version because we can correctly compare only source versions or encoded versions, but not a mixture of both. So, the case of low frequency gaps allows only marginal improvement by the target arithmetic encoding.

Looking at the underweight case from another perspective, let us assume that our encode vector contains only peak prefixes occurring with the same frequency and low frequency gaps situated between peaks. If we ignore all gaps as marginal contributors to the overall encoded length, then our peak-only encoding should be perfect. However, the number of symbols used in the peak-gap encoding is twice the number of symbols used in the peak-only encoding. This adds exactly one extra bit to the symbol binary representation. For realistic encode table sizes ranging from 2^{10} to 2^{15}, this extra bit comprises 7–10% overhead attributed to the low frequency separation gaps that are needed almost exclusively for order preservation. Since handling the discrepancies between substantial frequencies of peaks and gaps is the problem common to all dictionary compressions, *we estimate the overhead factor of order-preserving compression as roughly 10%.*

Both overweight cases (1) and (2) are poorly handled by Hu-Tucker encoding for the same reason. Very high frequency peaks in (1) would especially benefit from arithmetic supplement. There are real life indexes where one key value amounts to a half of all index key occurrences, e.g., NULL value of “male” in sex index. With the target arithmetic encoding, half of the keys can be reduced to three-five bits. On the other hand, in such cases, a single symbol of our dictionary-only encoding can encode a highly repetitive key, with no “stop” symbol as in arithmetic encoding, producing three-to-dozen bits of code. This brings us again to a relatively low impact of the target encoding on what can be achieved with the dictionary-only compression. The noise-level frequency variance in (4) offers a saving of approximately one bit per symbol if a target encoding is used.

Summarizing the effect of a supplementary target encoding on overall order-preserving dictionary-based compression, we see it as relatively low. We are certainly talking about the improvement expressed as a percentage, often at the order of 10%, whereas our experimental results, pre-

[2] Hu-Tucker codes are order-preserving binary Huffman codes. They hold prefix property. Due to the prefix property, a long sequence for infrequent gap can have a short sequence for only one neighboring frequent peak. Worse than that, the other side of the long Huffman code must decrease gradually to another short code, making it impossible to continuously interchange long and short codes, and thus damaging compression. A typical example of such combination is:

```
010
0110
0111
10      -   short code for frequent peak
11000   -   long code for infrequent gap
11001   -   this code for frequent peak should
            be short, but it is unavoidably long
1101
111
```

sented in Sect. 9, show five-fold compression achieved by applying the dictionary-only encodings to realistic data sets.

There still remains one more effectiveness parameter to assess: speed of the dictionary-only encoding. Dictionary encoding speed depends directly on the average source code size $\sum_1^K (len(q_i) \cdot f_i) / \sum_1^K f_i$. The longer the codes, the faster it goes. But, ignoring the constant $\sum_1^K f_i$, the sum $\sum_1^K (len(q_i) \cdot f_i)$ is exactly what we maximize while building individual encode vectors. Therefore, optimization of compression rate and speed coincide when producing each encode vector so that the speed optimization criterion needs to be factored only into the process of the best encode vector selection.

Comparing the speeds of order-preserving and order-indifferent dictionary encode procedures is largely an implementation-related issue. Here we only mention two conceptual-level differences: (1) order preservation requires up to twice as many symbols and thus needs up to one extra comparison to detect one prefix, and (2) order-preserving encoding needs no parsing optimization used in order-indifferent methods, thus avoiding a many-fold speed overhead.

Finally, one should note that neither greedy nor equalizing algorithms deliver optimal encodings. Our experiments with the equalizing algorithm demonstrated a five-fold compression rate on large realistic data sets and, at the same time, exhibited a two-fold noise factor when varying the weight W. Picking the best encodings from these variations produces a very significant shift toward the optimum, but room for better algorithms still remains. Compared to the published results on similar datasets produced by known traditional methods, our compression rate falls close enough to the average traditional algorithms rates to consider the largest portion of the optimal order-preserving compression already resolved.

8 Blending

Blending is a redistribution of the assigned frequencies in a string or string partition set aiming to either obliterate or substantiate some part of the set. In particular, gaps or end edges containing no frequency table strings must be substantiated with some non-zero frequencies if a chance of some string falling into the gap remains. This is a simple (first order) version of traditional blending that redistributes a small portion of the original frequencies into some under-weighted partitions. Note that in read-only databases, such as archived data collections or data stored on CDs, gaps with zero frequencies can be fully eliminated by excluding them from the set of trailing source substrings. By empty gap elimination, we can reduce the encode vector length up to two times.

In the original frequency table, if several strings extend string s, different encode partitions may include only a part of the extending strings, thus making application of the string s frequency ambiguous. A simple way to resolve this ambiguity – the way that also simplifies the dictionary building algorithm – is to recursively redistribute a string frequency among string extensions and delete the string until no extensions are left in the string set of the frequency table.

Assume that the encode vector is already built and string s and its extension s' belong to the original frequency table. During execution of encode procedure, if the selected source code q_i is a prefix of s, then q_i is a prefix of all s extensions, and, for the sake of this encoding instance, the pattern of redistributing frequencies from s to its extensions should not influence the decision on q_i selection done during a dictionary build. On the contrary, if in the same scenario q_i extends s and also stretches past the source string end, then ambiguity of picking q_i in step 2 of the position-independent encode procedure leaves room for a redistribution pattern to influence q_i selection and build a better dictionary encoder. The last (ambiguous) q_i encoded for a given source string is to be truncated at the final step 7 of decoding and, therefore, we want to use a few q_is as possible for the tail encoding in order to reduce the encode vector size. We achieve that by redistributing string s frequency into its most frequent extension, causing maximum imbalance of the redistributed frequency table.

Algorithm 3 (frequency table blending)

1. In the set S of all frequency table strings, find string s such that the set $\lfloor s \rfloor \cap S$ of all extensions of s in S is not empty and holds the prefix property. (String s detection can be organized into a single trie traverse.) If s is not found, terminate the algorithm.
2. Find s' in $\lfloor s \rfloor \cap S$ having the highest frequency, increase s' frequency by s frequency, and delete string s from the frequency table.
3. Go to step 1.

If source strings contain variable-length text padded with blanks to the common length L, the above blending would accumulate frequencies of all-blank substrings into the longest one, giving it a huge weight. This makes the greedy and equalizing algorithms deliver the following encode vector fragment (here stands for blank):

	q_i	symbol
edge		$i-1$
peak	$\ldots$	i
edge		$i+1$

In addition, our frequency table blending leads to the discovery of a combination of typical word endings with trailing blanks. For instance, substrings "ing" and "ed" are most likely to be among the peaks and will be encoded with a single symbol. Moreover, sequences of any frequent trailing characters like "*" or "_" are detected and encoded with a single symbol, one per each individual character.

Comparing it with zilch encoding, symbol i for peak "$\ldots$" is an analogue of the terminating symbol of Zandi et al. (1993). But at this point the similarity ends. In addition to a terminating symbol, zilch still has to assign explicit symbols for one, two, etc., blanks on each side of the longest trailing blank sequence. Zilch also produces similar chains of unnecessary symbols on each side of trailing "*", "_", "ing " and "ed ". This causes a significant compression rate suboptimality.

Our method provides the same advantage of a single-symbol encoding of frequent trailing patterns, compared to other compression algorithms described by Blasgen et al. (1993) and Antoshenkov et al. (1994). The secret here is that our encoding considers only strings of length L predetermined for a given source dataset. This allows a truncation of oversized trailing substrings, yielding a sufficiency of a single encoding symbol. All other algorithms deal with variable-length strings and miss this opportunity because, in their case, the stop rule must incorporate the source string length into the code itself. In our case, the string length is stored elsewhere (in a coding scheme), so that we simply need less information to encode.

Note that dealing with fixed-length strings does not reduce the generality of our method. We can always pick L to be bigger than any realistic string length and use padding characters for any desirable comparison rules. We pad with blanks for SQL comparison and pad with an artificial extra character lower than the lowest alphabet character when an extension of some string should compare high with the original string. Huge L and special padding are easily handled at implementation.

Frequency table blending is the only mechanism known today for automatic trailing symbol selection.

9 Experimental results

To compare the efficiency of our compression with other known methods and to see what kind of encode tables are actually built, we implemented a prototype software and ran several experiments on real-life data and on Wisconsin benchmark – widely used for comparing database performances.

In these experiments we used the dictionary-only version of our order-preserving compression, i.e., we skipped the target encodings [such as Hu-Tucker, arithmetic, or the work of Antoshenkov et al. (1994)] and used binary symbol representations instead. Encode vectors and corresponding common prefixes were obtained using equalizing algorithms. We ran our experiments on a VAX 6500 (Digital Equipment Corporation) machine during working hours with a normal development workload (trying to be as close to a production environment as possible).

The first real life dataset we tried was a collection of occurrences of global names present in all source modules of one of the Oracle Rdb (Oracle Corporation) components. There were 159 554 occurrences of the names, which we padded with blanks to the maximum length of 31 characters in order to enforce SQL comparison rules. A name occurrence dataset and an index built on it should be a backbone of any software development tool. In Table 3, a summary of compression characteristics for the global name occurrence dataset is presented.

We picked three encode vectors out of about one hundred vectors produced to exemplify compression characteristics depending on different encode vector sizes. The size of the prefix vector expresses a number of frequent text patterns and infrequent gap patterns (in B) to be replaced by symbols. So, the total amount of B needed to hold the encoding scheme is approximately *NumberOfIntervals*6+PrefixVectorSize*, assuming 4-B pointers to prefix string starts, 1 B to express a prefix length, and to specify interval type: peak, edge, or gap. The encoding scheme size depends on implementation and may vary within a 10–20% range.

At the high end, our method compresses 5 MB into under 1 MB of order-preserving code, yielding 5.422 compression rate and using extra 183 KB main memory B to hold the compression scheme. CPU time needed for compression/decompression is marginal compared to CPU cost of a single key insertion or retrieval. Compared to removing only trailing blanks by the run-length order-preserving compression (Antoshenkov et al. 1994), which is currently part of Oracle Rdb, and which yields 1.903 compression rate, we deliver extra 2.85 compression. This compression, applied to the same data stored along with data records, will yield the same five-fold compression or almost three-fold compression on top of the commonly used order-indifferent run-length compression of repeated characters.

If one wants to minimize the main memory area occupied by the compression scheme, one can pick the low-end scheme which takes only about 28 KB of main memory and still delivers a significant 3.388 compression factor, bringing 5 MB down to 1.5 MB.

For our next experiment we chose a collection of electronic mail similar to that used for text compression benchmarks. We extracted 666 666 word occurrences, converted them to uppercase, and padded them with blanks to the maximum size of 15 letters. The compression characteristics for this dataset are presented in Table 4.

Again, the equalizing algorithm manages to compress word occurrences put into a 15-character table attribute up to six times and to perform 2.668-fold extra compression on top of the order-preserving compression that removes trailing blanks. The traditional, order-indifferent compression methods, as summarized for five different plain source texts by Bell et al. (1990, Appendix B), have low compression rates of 1.59–1.77 delivered by Huffman and some flavors of Ziv-Limpel coding, and high compression rates of 3.02–3.56 delivered by “prediction by partial match”, dynamic Markov, and “WORD” compressions.

Since order-indifferent compressions of texts do not count any padding blanks, we should compare our padless 2.668 rate with the overall range of 1.59–3.56 produced by known order-indifferent methods. Our rate falls slightly above the traditional median of 2.575, and it would be fair to say that we achieved about the average rate of order-indifferent compressions. The fairness of this comparison is only slightly compromised by two unavoidable factors: order-indifferent compressions take advantage of frequent phrases stretching across word boundaries (this feature is deliberately excluded from the order-preserving setting) and, on the other end, order-preserving compression algorithms do several passes through the source or its random subset (whereas order-indifferent compression normally is restricted to one pass).

As we have already mentioned in the previous section, there remains room for compression rate improvement, which, in the future, may bring the order-preserving compression rate substantially closer to the best rates available today. For database applications, however, we should be looking at absolute rate six and count its relative improvements compared to other compression methods used

Table 3. Compression of global name occurrences. Here *Weight W* stands for a threshold above which a peak or edge with a product *PrefixLength*PrefixFrequency* should be included in the vector of encode intervals E. Also, *Compression rate* is a ratio *SourceDataSize/CompressedDataSize*, *(trailing blanks only)* shows a rate of compression which encodes only trailing blanks, *Number of intervals* is the encode vector E length (i.e., K), and *Size of prefix vector* is a total number of B occupied by prefixes in vector Q

Compression rate	Compression rate (trailing blanks only)	Number of E intervals	Size of prefix vector Q	Weight W
5.422	1.903	9204	128125	96
4.675	1.903	5242	71178	160
3.388	1.903	1464	18925	512

Table 4. Compression of email word occurrences. *Compression rate* is the *SourceDataSize/CompressedDataSize*, (*trailing blanks only*) is the compression which encodes only trailing blanks, *Number of intervals* is the encode vector E length, *Size of prefix vector* is a total number of B occupied by prefixes in vector Q, W is the threshold for *PrefixLength*PrefixFrequency* to be included in the encode vector E

Compression rate	Compression rate (trailing blanks only)	Number of E intervals	Size of prefix vector Q	Weight W
6.009	2.252	22752	197157	114
5.218	2.252	5147	40179	48
4.318	2.252	735	5142	384

in databases. Unfortunately, there are no database benchmarks that incorporate the effect of data compression into its metrics. On the contrary, a typical database benchmark attempts to avoid any compression impact.

For example, Wisconsin benchmark defines the content of its three text fields as picked from a set of strings "$*xxxxxxxxxxxxxxxxxxxxxxxxx*@*xxxxxxxxxxxxxxxxxxxxxxxx*#" where character variables $, @, and # take their values from the alphabet (*A*, *B*, ..., *V*) and cover all possible permutations of those. When we ran our equalizing compressor against the set of these 10 648 different strings, it took it 2 min to discover the underlying pattern, create a 74-interval encode vector, and compress the string set 11 886 times. Recall from Gray (1993) that in the benchmark relations there are three such text fields, comprising 75% of a relation space. With our compression automatically applied at creation of relations and indexes, we should look at 3–4 times smaller tables and some indexes shrunk many times compared to the benchmark authors' intention. In our view, future benchmarks should support compression measurements as an essential database feature, instead of attempting to generate uncompressable text.

10 Conclusion

We demonstrated here that the full power of dictionary encoding can be applied to order-preserving string compression. Common to all dictionary encodings, our encoding is very fast, but, being cast for order preservation, our method differs drastically from the traditional ones: no "learn while encode one long string" as in Lempel-Ziv, but rather search for commonality in a large number of short strings. For example, when compressing an ordered set of car license plate strings, one wins a great deal by detecting all-digit and all-letter position areas and working with them individually; one also wins by excluding unused letter combinations from the source string space. To utilize these potentials, we defined encode/decode procedures for position-dependent, position-independent, and mixed cases, and also considered arbitrary restrictions by dealing with any set of *L*-long strings.

Aiming at the optimal compression rate, we uncovered the underlying structure of compression-suitable encoding as a set of peaks surrounded by edges and separated by gaps [this structure yields better compression than zilch symbols (Zandi et al. 1993)]. Then we explored probabilistic (frequency-based) optimality and found that frequency balancing in the encoder leads toward the optimal dictionary selection and that the original frequency table disbalancing helps to determine optimal trailing patterns. These findings are incorporated in the dictionary building and frequency blending algorithms. It turns out that encoding speed optimization is closely related to optimization of the compression rate, hence the above algorithms resolve both tasks simultaneously.

Order-preserving compression based on dictionary/arithmetic pair matches similar order-indifferent compression, delivering the identical compression rate with a limited number of extra partitions. This suggests that in database systems, many or all table fields can be compressed without losing the order, and then used without decompression for comparison in select, sort, merge join, and index B-trees, improving storage utilization and operational speed, provided that, for join, fields from common domains share encode vectors.

References

1. Antoshenkov G, Lomet D, Murray J (1994) Order-preserving key compression. DEC Cambridge Research Laboratory, Tech Rep, July
2. Baer J, Lin Y (1989) Improving quicksort performance with codeword data structure. IEEE Trans Software Eng 15: 622–631

3. Bayer R, Unterauer K (1977) Prefix B-trees. ACM Trans Database Syst 2: 11–26
4. Bell TC, Cleary JC, Witten IH (1990) Text Compression (Adv. Ref. series). Prentice-Hall, Englewood Cliffs, NJ
5. Blasgen MW, Casey RG, Eswaran KP (1977) An encoding method for multifield sorting and indexing. Commun ACM 20:874–878
6. Elias P (1975) Universal codeword sets and representations of the integers. IEEE Trans Inf Theory, 21: 194–203
7. Graefe G (1993) Query evaluation techniques for large databases. ACM Comput Surv 25: 73–170
8. Gray J (ed) (1993) The benchmark handbook for database and transaction processing systems, 2nd edn. Morgan Kaufmann, San Mateo, Calif
9. Hu TC, Tucker AC (1971) Optimal Computer Search Trees and Variable-Length Alphabetical Codes. SIAM J Appl Math 21:514–532
10. Huffman DA (1952) A method for the construction of minimum-redundancy codes. Proc IERE 40:1098–1101
11. Moffat A, Zobel J (1992) Coding for compression in full-text retrieval systems. Proceedings of Data Compression Conference, Snowbird, UT, pp 72–81
12. Zandi A, Iyer B, Langdon G (1993) Sort order preserving data compression for extended alphabets. Proceedings of Data Compression Conference, Snowbird, UT
13. Ziv J, Lempel A (1978) Compression of individual sequences via variable-rate coding. IEEE Trans Inf Theory 24:530–536