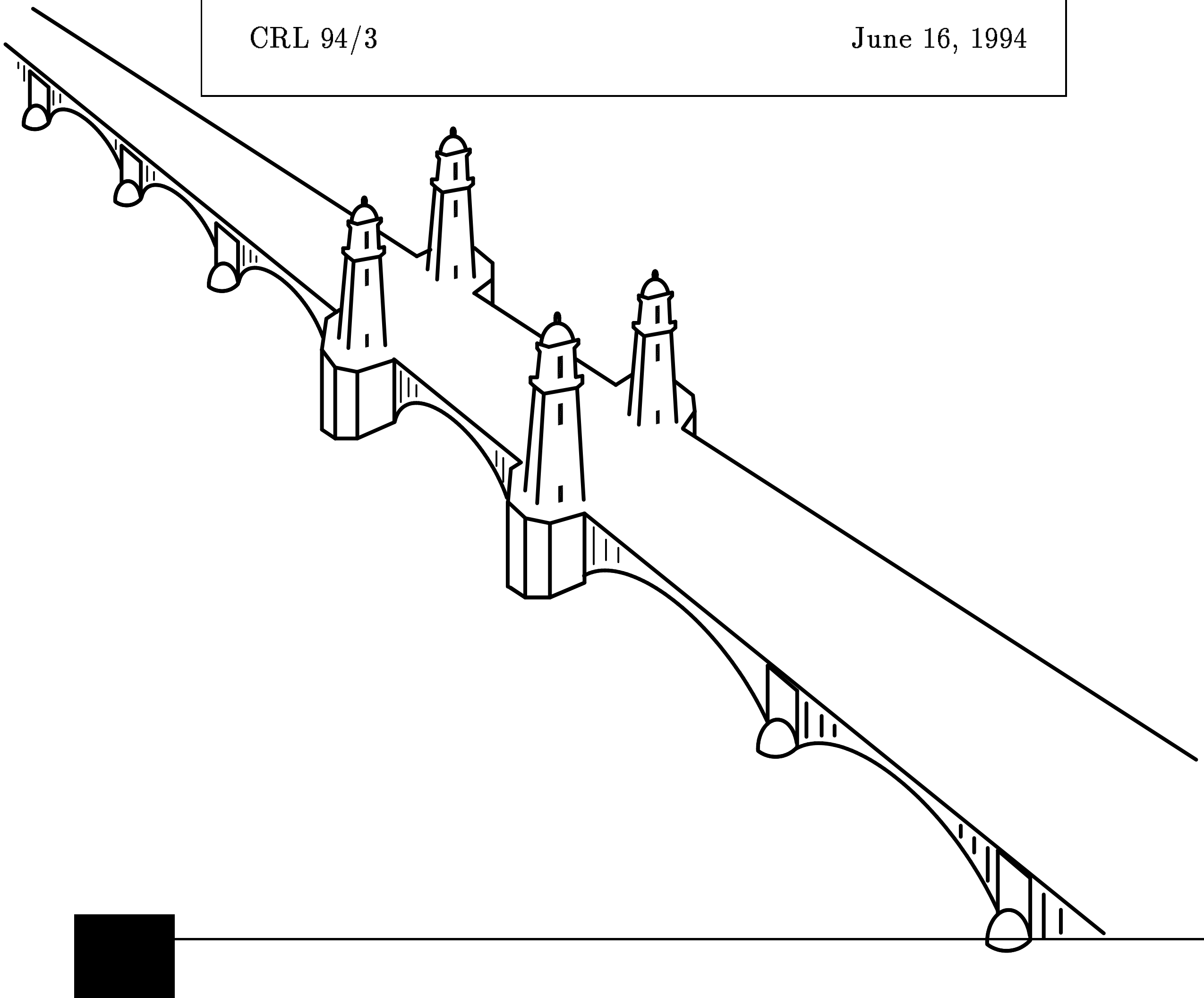
Order Preserving Key Compression

Gennady Antoshenkov David Lomet
James Murray

Digital Equipment Corporation
Cambridge Research Lab

CRL 94/3 June 16, 1994



CAMBRIDGE RESEARCH LABORATORY
Technical Report Series

Digital Equipment Corporation has four research facilities: the Systems Research Center and the Western Research Laboratory, both in Palo Alto, California; the Paris Research Laboratory, in Paris; and the Cambridge Research Laboratory, in Cambridge, Massachusetts.

The Cambridge laboratory became operational in 1988 and is located at One Kendall Square, near MIT. CRL engages in computing research to extend the state of the computing art in areas likely to be important to Digital and its customers in future years. CRL's main focus is applications technology; that is, the creation of knowledge and tools useful for the preparation of important classes of applications.

CRL Technical Reports can be ordered by electronic mail. To receive instructions, send a message to one of the following addresses, with the word **help** in the Subject line:

On Digital's EASYnet:	CRL::TECHREPORTS
On the Internet:	techreports@crl.dec.com

Cambridge Research Laboratory
One Kendall Square
Cambridge, Massachusetts 02139

Order Preserving Key Compression

Gennady Antoshenkov David Lomet
James Murray

Digital Equipment Corporation
Cambridge Research Lab

CRL 94/3 June 16, 1994

Abstract

Order preserving data compression can improve sorting and searching performance, and hence the performance of database systems. We describe a new parsing (tokenization) technique that can be effectively applied to "keys", producing substantial compression. It can be applied to both the uncompressed and compressed strings, permiting a more flexible choice of variable length dictionary entries and their compressed forms. The key notion is that the way that entries partition the space of strings determines how the entries are to be parsed. The result is economy in the number of encodings that are needed in order to preserve order and great flexibility in how they are encoded. We illustrate one use of order preserving compression, padding character compression for multifield keys. This demonstrates the dramatic gains possible with the new technique, some of which cannot be achieved by any other method.

Keywords: compression, order preserving, padding, multifield, searching, sorting

1 Introduction

There are many techniques that permit the lossless compression of data. Most of these compression techniques do not, however, preserve the ordering of the compressed data [5, 2]. There are important reasons for wanting data compression that preserves order. Such a compression technique facilitates both sorting and searching.

Sorting: Tag sorting is a method by which, instead of moving entire records when sorting them, one extracts the sort key (the tag) of each record and stores it with a record pointer [6]. This reduces greatly the amount of data that needs to be moved during sorting and improves cache locality during comparisons and moves. AlphaSort [8], currently the world's fastest sort, exploits tag sort in this way. Order preserving key compression reduces the size of the tags, speeds comparisons and moves, and further benefits cache locality. There is an especially large payoff for large tags and multifield tags.

Searching: The index terms in an index tree, e.g. a B-tree [1], can store the order preserved compressed keys instead of the uncompressed variants. This permits a binary search within the index node, while facilitating increased fanout. This is particularly important when dealing with multifield keys [3]. Without being able to eliminate the storage used by the pad characters, the fixed size keys must be represented in full. Such multifield keys can destroy index node fanout. This leads to increased index tree height and reduced search performance, as well as greatly increasing the storage consumed by each index.

For both of these compression tasks, it is important that the compression technique be static as well as order preserving. Only a static technique will preserve the order over time, which is important in both these applications. This rules out many of the most powerful techniques, such as those based on the Ziv-Lempel method [10], which are more attuned to compression of long text passages in any event. Also, the need to preserve order eliminates many dictionary techniques such as Huffman's [5]. As we shall see, however, our new approach can be applied to non-order preserving situations as well, permitting us to potentially do better than Huffman coding.

1.1 Order Preserving Compression

There are many special case order preserving compressions. For example, the characters represented by bytes might not be dense, e.g. the byte can encode more characters than are needed. Alphabetic characters might need only 26 distinct bit patterns, perhaps up to 64 if one includes upper and lower case and some punctuation. A byte can encode 256 patterns in its eight bits, while 64 patterns requires only six bits, which if used for the representation, would compress the data by 25%. This may be effective in some cases, but this compression is not large, nor can it be exploited as a general method.

1.1.1 Arithmetic Compression

Arithmetic compression can be used for order preserving string compression. It is based on knowing the probabilities of the items to be encoded, and works by adding cumulative probabilities to the results of prior encodings that have already been calculated. This calculation preserves ordering when the cumulative probabilities are computed based on entries in sort order [2].

Two factors work against using arithmetic coding.

- Especially for tagged sorts, where every sort key needs compression during the run time of the sort operation, one needs the compression algorithm to work at very high speed. Dictionary approaches, especially those that can deal with units of multiple symbols, are much faster than arithmetic methods. [1]
- A dictionary approach can be very closely tailored to the problem. For example, one might compress some entries without compressing others. Indeed, we illustrate this in our running example.

Finally, arithmetic coding, while optimal given the right model of the input, does not show much improvement over dictionary methods when the frequencies of dictionary entries are all small. Hence, it is hard to justify its increased computational cost.

---

[1] In [7], arithmetic coding was reported as being a factor of 40 slower than their dictionary approach.

1.1.2 Dictionary Compression Methods

One systematic and "optimal" method of performing order preserving compression is the Hu-Tucker algorithm [5]. Like Huffman coding, it builds an optimal weighted binary tree, where the weights are assigned based on the frequency of the entry to be encoded. The entries constitute a "dictionary" of tokens to be compressed. Unlike Huffman encodings, the weighted nodes cannot be re-arranged arbitrarily because the order of the compressed forms needs to be the same as the order of the original entries. Like Huffman's method, the compressed forms are of variable size and have the prefix property in which no form is a prefix of any other.

Hu-Tucker has two limitations also.

- Hu-Tucker does not address the issue of how to parse the input string. This problem needs to be solved so as to permit correct ordering of dictionary entries. In particular, how does one order entries when one entry is a prefix of a second entry?
- The set of compressed forms is required to have the prefix property by the limitations in decoder parsing. It is important both for effectiveness of compression and for flexibility that we not require the prefix property for the encoded forms.

1.2 Our Approach

Our method, which we shall refer to as the ALM method for obvious reasons, is a dictionary method in which the parsing for encoding and decoding can be performed by the same parser. And, importantly, the parser does not require that dictionary entries have the prefix property. Thus, ALM permits very long strings to be compressed very flexibly. The parsing method relies on the ordering of the dictionary entries. Order preservation is assured by ordering encoded forms in the same order as the dictionary entries. Ordering of entries may, however, require information about strings that follow the entries.

ALM's parsing method permits us to reduce the number of encodings needed to preserve order compared with its only real competitor, the ZIL method [9] which we learned of during the writing of this paper. ALM's very flexible parsing also permits more choice of compressed forms, leading to potentially higher compression. We briefly compare ZIL to ALM in section 4.

1.3 Multifield Compression

1.3.1 The Problem

In many database systems, and in particular, in the ISO standard SQL [4], only fixed length fields are supported. Thus, when one has, for example, a NAME field in an EMPLOYEE relation, the size of the field must be specified as sufficiently large to accommodate the names of all employees. Typically, and surely in this case, the average size of an employee's name is much shorter. It is not unusual to have a NAME field of 50 to 100 bytes while the typical employee name is perhaps 15 to 20 bytes.

The SQL standard is very precise as to what is required when fields of different lengths are compared. One extends the shorter field with blanks ('20'X) until the shorter field is of the same length as the longer field and then one does the comparison as between two fields of the same length. The correct result can then typically be achieved by using the standard hardware supported byte string comparison. Thus:

“xyz” :: “xyzuvw” *means* “xyz ⊔⊔⊔” **::** “xyzuvw”

where ⊔ denotes a blank and **::** signifies comparison. Representing strings using ⊔ as our symbol for a blank will be our consistent practice throughout this paper.)

Multifield comparisons require that every component field be padded out to its declared length. Thus, we have

A || B :: A' || B' (where || denotes concatenation) *means*
val(A) || “ ⊔⊔⊔” || val(B) || “⊔⊔” :: val(A') || “ ⊔⊔⊔⊔” || val(B') || “ ⊔”

where the different number of pad characters are intended to denote that the relevant values are of different lengths, all shorter than the declared lengths of A or B.

ALM works well in solving the multifield compression problem. We use this problem as a running example to illustrate the power of the ALM approach.

1.3.2 Prior Work

Multifield comparison was recognized as a problem during the System R project at IBM and an IBM technical report [3] describes their solution. Briefly, they propose that padding characters can be truncated if one inserts control characters at intervals of N characters, where N is a fixed parameter, into the resulting concatenated character string. The control characters indicate whether the next field begins or the current field continues. Since the control characters are at fixed intervals, a byte string compare will always compare one control character with another. The control character that starts a new field always compares low to the control character for a field continuation. Values are padded out to an integral multiple of N, which means that fields always begin immediately following a control character.

The specific control characters used were the following:

‘FF’X : denotes that the preceding N bytes contain no padding characters and that the field is continuing.

‘nn’X : denotes that the previous field ends within the preceding N bytes with ‘nn’X denoting, in hexadecimal, how many non-padding characters there were in the preceding N bytes of the representation.

In [3], an example of the representation of a key consisting of the two fields with values “ABCDEF” and “XYZ” is the following, with N = 4:

```
‘‘ABCD"||‘FF’X ||‘‘EF"||‘0000’X||‘02’X||‘‘XYZ"||‘00’X||‘03’X
           |               |       |              |     |
           Control         Pad     Control        Pad   Control
```

For this method, continuing fields always compare high to fields that are terminating. Further, if both fields are terminating, the longer one compares high. Unfortunately, this is **not** the way that comparison is specified in SQL. It is not the same as extending the fields with blanks, which have a hexadecimal representation of ‘20’X.

1.3.3 Impossibility of Simple Concatenation

In [3], it was shown that multifield comparison cannot be solved with an encoding that involves a straightforward concatenation of variable length strings, no matter what information is appended onto the end of each component string. The problem is that any end markers may also occur within the strings and hence our control characters are compared with data characters. This proof is correct as it is written! The conclusion drawn is that "an encoder must mutilate the strings in some way" to correctly preserve order.

We can do much better by viewing the multifield string problem as an opportunity to apply order preserving compression. The "mutilation" then merely consists of applying the same order preserving compression to all instances of padding characters (end markers), be they at the end of a string or elsewhere in the string.

2 Forward Context Parsing

Solving the multifield string problem by order preserving compression requires that compression be applied to strings of padding characters of many different lengths. That is, we need to encode strings of one pad character, two pad characters, ... n pad characters, where clearly, the shorter strings of padding characters are prefixes of longer strings. This is an example of a general problem with dictionary compression techniques of encoding entries which do not have the prefix property.

Consider compressing the pad character strings "␣␣" and "␣␣␣" such that order is preserved when all other characters are encoded via the identity mapping and are unchanged. It should be clear that for the other characters, the encoding is order preserving. But how do we choose an encoding for these strings that preserves order. Assume that the encodings for "␣␣" and "␣␣␣" are e(␣␣) and e(␣␣␣) and that these encodings have the prefix property so that they can be uniquely ordered. Suppose that e(␣␣) > e(␣␣␣). Then, when "A" compares low to blank, the string "␣␣A" becomes "e(␣␣)A", which is greater than e(␣␣␣). Hence order is not preserved. Now suppose e(␣␣) < e(␣␣␣). Then when "C" compares high to blank, "␣␣C" becomes "e(␣␣)C" which is less than e(␣␣␣), which likewise does not preserve order. Hence, any such scheme which provides only one encoded form for a string and permits the order to be determined as described cannot succeed.

Notice that an identity transformation trivially preserves order. This results in different length strings encoding to strings of these different lengths. That encoding does not have a unique ordering between "␣␣" and "␣␣␣".

Rather, how these strings compare is a property of what comes after the shorter string. With this observation in mind, let us now try to solve the padding problem.

2.1 Solving the Padding Problem

We would like to compress padding characters into some representation that preserves order for many lengths of padding character strings and that usually results in a dramatic reduction in the length of a string. Our prior observation leads to the following insight. We need different encodings for a given length string of padding characters depending on whether the character that follows the string compares high or low to the padding character itself. That way, for example, a "␣␣" would be encoded differently when followed by byte that compares low to blank than when followed by a byte that compares high.

Since it must be possible to decode the encoded string and recover the original string, our encodings must represent, in some way, the lengths of the strings they encode. Further, recall, we need two representations for each length (except for the longest string that we encode, which is not a prefix of a longer string), one for the context in which the following character compares low to the pad character, and one for when it compares high. The following is a solution to the padding problem. Encode a string of blanks as

$$\text{␣} \;\|\; \text{length(string)}$$

when the character following the string compares low to a blank. Encode this same string as

$$\text{␣} \;\|\; (2^*\text{maxstring - length(string)})$$

when the character following the string compares high to a blank. It is straightforward that this encoding is order preserving. Short strings compare low to longer strings when the following character is low and compare high to longer strings when the following character is high.

Thus, all strings of padding characters with lengths between 1 and 128 can be represented in two bytes. All pad strings are compressed except for strings of length one which now need two bytes, not one, and those of length two which remain unchanged.

Pad Strings	Encoding	Trailing Context
␣	␣1	character comparing low to blank
␣␣	␣2	"
␣␣␣	␣3	"
␣␣␣␣	␣4	all contexts, only one encoding needed
␣␣␣	␣5	character comparing high to blank
␣␣	␣6	"
␣	␣7	"

Table 1: The pyramid formed when multiple length blank strings are to be parsed so as to preserve order. ␣ is used to denote a blank.

2.2 Identifying and Ordering Strings

The above technique is a special case of a more general approach which involves carefully identifying when strings need multiple encodings. One first identifies all strings for which encodings are desired. Those strings are organized so as to partition the entire range of string values into disjoint subranges which are correctly ordered. This requires that strings that are prefixes of other strings have multiple encodings.

A simple example of how this is done can be seen in the padding example. Let the maximum size string to be encoded be of length four. Then the ordering that is needed is given in the Table 1. The pyramid is formed by the need for the shorter(prefix) string to appear both before and after the longer string(of which it is a prefix). [This kind of parsing is also permitted by the ZIL method. ALM parsing, as will be seen in section 4, includes the ZIL capability as a special case but is strictly more flexible.]

2.3 Strings as Boundaries and Prefixes of Ranges

2.3.1 The Nature of the Problem

We want to generalize from the above examples by applying context dependent encoding to the general order preserving compression problem. There are two aspects to the order preserving translation problem.

- The input string must be parsed to yield tokens, which are the units to be encoded.
- The tokens must be mapped(encoded) so that order is preserved.

The method we are about to describe introduces new techniques for both of these tasks. The input string is parsed to yield tokens by taking "trailing" context into account, not just relying on exact match of a string prefix with the substring denoting the token. In addition, this "trailing" context can be used to change the mapping of the token. There is no requirement that a given token be mapped in only one way.

There are usually two additional requirements that must be satisfied if order preserving translation is to be used for compression.

- all input strings must be encodable. This is called dictionary completeness in [2].
- encoded strings must be decodable so as to produce the original input string.

2.3.2 The Dictionary

To perform this context dependent parsing and translation, we arrange the entries of our dictionary into an ordered list such that the list decomposes the entire string space into disjoint ranges. Each range must have a unique prefix that is one of the tokens on our list. This prefix is consumed by the encoding process.

Not all dictionaries satisfy the needs of our encoding process. Consider the set of strings consisting only of "A", "B", and "C". Let us try to compress sequences of three "A"s and three "B"s. So we construct the table consisting only of "AAA" and "BBB" as indicated in column 1 of Table 2. These entries are given encodings that are ordered by the position that our entries have in the table.

This listing of encodings is clearly inadequate. There is no way, for example to translate the strings such as "AB", or "A", "B", or "C" for that matter, etc. We need to make sure that all possible strings have encodings. One way to do that is to include all individual characters in the table of strings to be encoded. One possible result is given in column 2 of Table 2.

1	**2**	**3**	**Range**	**Seq. No.**
	A	A	[A-,AAA-)	0
AAA	AAA	AAA	[AAA-,AAA+]	1
		A	(AAA+,A-) [A-,AB-)	2
		AB	[AB-,AC-) = [AB-,AB+]	3
		AC	[AC-,B-) = [AC-,AC+]	4
	B	B	[B-,BB-)	5
	BB	BB	[BB-,BBB-)	6
BBB	BBB	BBB	[BBB-,BBB+]	7
	B	B	(BBB+,B-) [B-,C-) = [B-,B+]	8
	C	C	(B+,C+] = [C-,C+]	9

Table 2: Strings consisting of characters $\{A, B, C\}$ to be encoded in an order preserving way. The pluses and minuses represent respectively the highest and lowest possible string values.

Now all substrings of length one have translations, and for context independent encodings, that would be sufficient to encode all strings. However, we wish to encode "AAA" and "BBB" as well. With the entries of column 2, we can encode all prefixes preceding and including "AAA" and also prefixes following and including "BBB". However, we cannot translate strings that order between "AAA" and "B", for example "AB". We need entries between "AAA" and "BBB" to take care of all possibilities between these entries. One example is given in column 3 of Table 2.

Table 2 illustrates the constraints that must be satisfied to successfully encode strings in an order preserving way. The tokenization of the input string requires the following:

- Each range in our table must have as a common prefix one of the entries on our list.
    1. An entry that is not a prefix of a longer entry is the prefix for a range that includes all strings for which it is a prefix.
    2. When one entry is a prefix of a longer adjacent entry, the prefix for the range bounded by these entries will be this shorter entry.

- Any string is in exactly one range, and the ranges partition the set of all strings to be encoded.

All strings fall into exactly one range of the partitioning and the common prefix for that range is the token that is encoded. Thus, all strings can be encoded unambiguously. The entries of column 3 partition the set of all strings and each range of the partition has a prefix that is one of the entries of the column.

Two observations illustrate the flexibility that is permitted in choosing which dictionary entries.

1. When a multicharacter string is listed, not all of its prefixes need have encodings. Thus, "AA" has no explicit encoding, despite the fact that both "A" and "AAA" do.

2. Symmetry about some encoded multicharacter string is not required. Thus, "BB" is encoded when it precedes "BBB", i.e. when it is either the end of the string or is followed by an "A". But it is not encoded as a distinct entry when following "BBB". There, it is encoded by applying the encoding for "B" twice.

2.3.3 Traditional Parsing Techniques

The traditional parsing schemes are context independent and are special cases of the context dependent parsing described here. For context independent parsing, the prefix property is satisfied and each entry defines a range that includes all strings of which it is a prefix. Our parsing method deals with entries satisfying the prefix property as a special case. However, we also permit entries that can be prefixes of other entries.

There are existing techniques for parsing strings for compression using dictionaries of variable size entries which do not have the prefix property. One can do "optimal" parsing in which all possible parsings are examined, and where the shortest resulting encoded form is selected. This is not typically done because of the computation cost. More typically, one does greedy parsing in which the largest matching dictionary entry is the one selected for encoding. Neither of these two strategies can guaranteed that order is preserved.

```
    let In  be the input string to be translated
    let Out be the output string, initially empty,
               that will contain the translated input string
    let D[] be the table of decoded entries (to be translated)
    let E[] be the table of their translations(encodings)

  do while( In ~= empty)
    i <- Search(D, In)  % D[i] is "matching" entry
    In <- Truncation(D[i],In)
    Out <- Out||E[i]
    end do while

  return(Out)
```

Figure 1: The conceptual parsing and translation procedure.

One can also handle variable length dictionary entries by extending the entries with enough trailing context to ensure that no entry is a prefix of another. If a given prefix is extended by an additional character, then all strings that include that prefix followed by any character must be included in the table. Thus, if "AA" is included in the encoding table, then so must "AB", "AC", etc. This explodes the size of the dictionary and decreases the effectiveness of the encoding. Of course, when dealing with bit strings, one has a two character alphabet and the "explosion" is very well contained. For larger alphabets, context dependent encodings provide real leverage because they avoid this explosion.

2.4 The Conceptual Parsing Function

Conceptually, the translation process is very simple. The procedure, in its abstract form is given in Figure 1.

There are, of course, many data structures and search techniques that might be used in the translation process. Abstractly one searches for a dictionary entry that matches the prefix of the remaining string and for which the remaining string falls within its associated range, as indicated in, e.g.,

Table 2. The dictionary is represented by the vector D[] in the procedure. of Figure 2.

3 Forward Context Encodings

What we have described so far is the parsing of the original string and the substitution of parsed input tokens from our dictionary by some translated form. This is half the task. The other half is chosing translated forms for the tokens so that decoding is possible, and the decoding process itself. Decoding has the constraint, of course, that it must regenerate the original input string. We exploit multiple "context dependent" encodings for some substrings in performing our order preserving compression. However, unlike encoding, a given substring must have only one translation when we are decoding. We tackle encoded forms and their decoding here.

3.1 Prefix Property Encoding

The simplest form of encoding is one in which we can totally order the encodings. When the ordering of the unencoded substrings, as enumerated say in Table 2, column 3, matches the ordering of their encoded forms, then order is preserved. We begin by requiring that the encoded forms be capable of being ordered among themselves with no further context. Thus, we can order the encodings by simply sorting them. This is possible exactly if the encoded forms satisfy the prefix properly.

The simplest possible encoding to understand is derived by simply using the sequence number of the substring to be encoded. Thus, in the example of Table 2, the numbers from zero to nine can be used. We assume here that all sequence numbers are represented in the same number of bytes. A given substring may need to appear more than once in the list of substrings to be encoded based on its following context. This means that there may not be a one-one correspondence between encodings and substrings encoded, which is illustrated in Table 2.

Huffman or Hu-Tucker based encodings are tree addresses of leaves in optimal trees. The average length of the paths in these "weighted" trees is minimized based on the frequency with which its unencoded character appears. The bit strings that are the encoded forms are variable in length but have the prefix property. Hence, in decoding the string of encodings, it is straightforward to identify (parse) this string and map the encoding back to the unencoded substring. This can be done via a tree walk or, equivalently, by performing a binary search of a vector containing the encodings, searching for the encoding that matches the prefix of the string being decoded.

3.2 Symmetry with Parsing Input Strings

Despite the great generality permitted by our translation process, if we confine ourselves to generating encodings with the prefix property, we have not captured all order preserving translations. In particular, there is a trivial order preserving translation that is not included. Consider the example in table 1, and let the substrings identified for encodings be as given in column 3. Then, what we have described does not permit us to use the identity mapping for these substrings, despite the fact that this is clearly order preserving since the original string is unchanged. The crucial point here is that the variable length encodings do not have the prefix property. It is impossible to construct a well defined ordering for encoded strings when one encoded string is a prefix of another without making an assumption about the string following the substring being encoded. This is, of course, a similar problem as that encountered by the unencoded forms.

3.3 Better Pad Compression

The above discussion is not merely theoretical. There are good reasons for wanting the added flexibility described above. Consider yet again the multifield pad compression problem. Suppose that a multiword text string can be a value for one of these fields where a blank character is used to separate the words. We use blank is the pad character as well.

The situation we have constructed results in singleton blanks occurring with some frequency. In the encoding we described above, all strings of blanks, including such singletons, require two bytes to represent them, a blank followed by a representation for the length of the string. So, all singleton blank strings are doubled in length. While there may be enough longer blank strings for us to achieve an overall reduction in the size of the entire multifield string, we can probably do better. Consider the following blank string frequencies:

```
singleton blank: .75
longer blank string: .25; average length = 10 characters
average blank string: .75*1 + .25*10 = 3.25
```

Then using our original encoding, we get an average blank string of

```
.75*2 + .25*2 = 2.00
```

That is, the average blank string, which before was 3.25 bytes long, will be reduced by 1.25 bytes, and now be 2 bytes long.

We can do better than this if we do not compress singletons, but only the longer strings, even if we need to use a longer encoded form for these long strings. Suppose that this required us to make the longer strings three bytes. Then

```
.75*1 + .25*3 = 1.5
```

Thus, we gain an extra .5 bytes per string savings in the encoded strings.

One order preserving encoding that permits the above is to translate singleton blanks to singleton blanks and all longer strings to

␣␣ || length(string)

when ␣␣ is followed by substrings that compare low to blank and

␣␣ || (2*maxlength - length(string))

when ␣␣ is followed by substrings that compare high to blank.

The question is, why is this effective? It is so exactly because the variable length encodings can be ordered correctly when the context, i.e. the substrings that can follow them, are considered, **even though they cannot be ordered effectively when just considering the substrings themselves**.

3.4 Ordering the Encodings

We need to understand the correct way to order encodings that do not have the prefix property and hence require that the following substring be examined. When we have two encodings, where one is a prefix of the other, it is impossible to determine which should come first without making an assumption about what follows the shorter string. The assumption that is usually implicitly made for comparison purposes is that the shorter string is followed by binary zeros, and hence that the longer string always compares high to the shorter string. But when a substring is embedded in a longer string its following substring can potentially be many other strings. Hence, we need to order substrings that we wish to use as follows:

Let str1 immediately precede str2 in the ordering of unencoded strings as previously described. Note that str1 need not be a prefix of str2. Further, let HIGH(str) be the highest substring that can follow str and LOW(str) be the lowest such value.

First, assume that encode(str1) is a prefix of encode(str2). Then we require that

$$\mathrm{encode(str1)}\ \|\ (\mathrm{encode(HIGH(str1))} < \mathrm{encode(str2)}$$

Next, assume that encode(str2) is a prefix of encode(str1). Then we require that

$$\mathrm{encode(str2)}\ \|\ (\mathrm{encode(LOW(str2))} > \mathrm{encode(str1)}$$

where the comparisons are done out to the full length of the longer string if that is needed. It is not necessary to compute HIGH(str) or LOW(str) out further the the length of the longer string as the comparison must distinguish the ordering by then.

We are accustomed to thinking of HIGH(str1) as the highest possible string value and similarly, of LOW(str1) as the lowest possible string value, independent of context. But that is not the case when the strings being encoded do not have the prefix property. As an example, when dealing with the encoding of padding blanks (the pyramid example), the string ␣␣ occurs twice in the pyramid. The set of strings that follow it in each context are not the same. It is the context dependent HIGH or LOW string that we use in the ordering.

The ordering that we need for encoded values is, in fact, exactly the same ordering property that we need for the unencoded values. There is a complete symmetry in what is required for these two tables of values. And exactly the same translation algorithm is used for encoding and decoding. The only difference is that the roles of the two tables are interchanged.

3.5 Equivalence of Encoding and Decoding

What we have achieved is order preserving translations in both directions, where the strings encoded **or** decoded can be variable length and can be prefixes of adjacent entries. Indeed, we do not care whether we are encoding or decoding. The procedure for translation is the same as is given in section 2.4. The translation produced by that procedure is unique. However, a given substring might be translated differently depending on what follows it. This is content dependent translation.

When encoding, the usual assumption is that any entry being encoded can follow any other entry. Decoding works from strings produced by prior encoding. Hence, for decoding, the set of strings that can follow an entry may be restricted. However, there is no inherent reason why the entries to be encoded cannot be context dependent. For example, in English, a ‘q” must be followed by a ‘u”. Context dependence permits more effective compression.

The correctness of the method relies on the facts that

1. the tokenization is unique (there is no choice in the procedure)
2. the tokenization is complete (tokenizing all input strings),
3. the encoded forms are ordered exactly as the unencoded forms. The translation table D[] is ordered and the encodings in E[] are ordered in the same way (when context is considered).

The fact that the decoding satisfies the same constraints permits it to use the same translation procedure, executed with encodings and decodings reversed. Thus, the translation process is run in reverse to perform the decoding to re-create the original string.

4 Comparisons with Other Methods

4.1 Improving on Hu-Tucker

Traditional Huffman and Hu-Tucker encodings are produced from optimal binary trees, where the tree addresses are the compressed encodings. The trees are optimal in that the “weighted” lengths of the paths to the leaves is minimized. By permitting encodings that do not satisfy the prefix property,

Frequency	Hu-Tucker	ALM
.49	00	0
.01	01	011
.49	10	10
.01	11	1011
Average(Hu-Tucker) = 2.0		
Average(ALM) = .49*1 + .01*3 + .49*2 + .01*4 = 1.54		

Table 3: Comparison of Hu-Tucker coding with ALM coding. Hu-Tucker requires its encoded forms to have the prefix property, while ALM does not.

it is possible to construct encodings that are more effective at compressing data than Hu-Tucker. Consider the example of Figure 2, where we have four entries to be encoded and we compare ALM with Hu-Tucker coding.

The entry frequencies that are given prevent Hu-Tucker from assigning a bit string of length less than two to any encoding. ALM is not so constrained. While ALM cannot assign a '1'B as the encoding for the third entry, it can assign a '0'B for the first entry. A '1'B cannot be assigned for entry three because the suffix appended to entry one to form entry two must be greater than all possible continuations produced by entries one through four. This suffix is '11'B, and no entry starts with a prefix that is greater than '10'B.

One can consider ALM in terms of binary trees, in a way analogous to what Huffman and Hu-Tucker exploit. The difference is that ALM does not require that only the leaf nodes represent valid encodings. It is certainly true that not all interior nodes of a binary tree can be used as encodings, but we can find some that are acceptable. For interior nodes, we also need to know whether the token so identified is less than, greater than, or both (i.e. the bit pattern represents two ranges). Applying this observation to our example gives us the trees of Figure 2.

4.2 Improving on ZIL

ZIL [9] is described as a parsing method based on an augmented trie. We map our method to an augmented trie to illustrate the difference. First, we present an example that will serve to demonstrate the ALM generalization

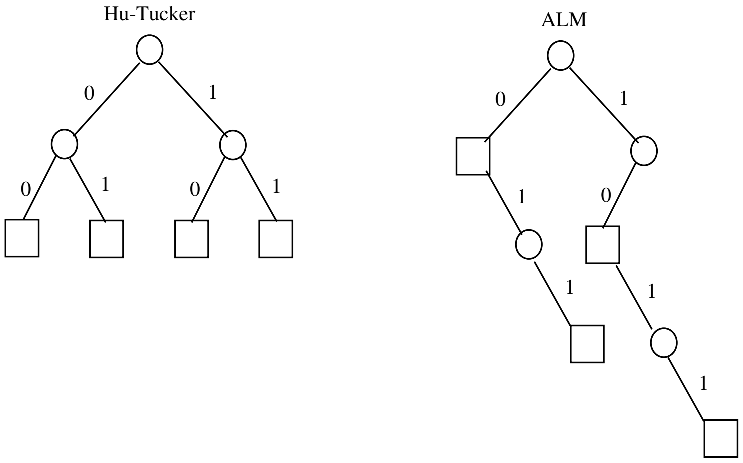


Figure 2: The binary trees representing the encoded entries from the encodings represent in Table 3. The boxes indicate the nodes that denote encoded entries, the circles the other interior nodes.

with respect to ZIL.

4.2.1 More Pad Compressions

There is no need for order preserving translations to be symmetric about the longest string being translated, as was indicated in section 3.3. We only need do that when we wish to provide a translation for the shorter string in both contexts.

Using again our padding compression example, we note that very few byte encodings precede '20'X representing a blank which is used as the padding character. We do not expect to frequently encode a string of blanks followed by a character that compares low to blanks. Thus, we might want to conserve our translations of blanks to provide better encodings for the blank strings that are followed by characters that compare high to blanks. We can do that by arranging our translation table as follows:

Here we assume that we can only store numbers up to 7 in the byte that follows the blanks. In reality, we can store numbers up to 255 in a byte. What the asymmetric entries then permit is for us to compress longer strings of blanks than when the entries are symmetric. However, these encodings work only when the blank string is followed by characters that compare high to

Padding Strings	Encodings
␣	␣
␣␣␣␣␣␣␣␣	␣␣ ‖ 0
␣␣␣␣␣␣␣	␣␣ ‖ 1
...	
␣␣␣	␣␣ ‖ 6
␣␣	␣␣ ‖ 7
␣	␣

Table 4: Encoding of strings does not require symmetry about the longest string. Symmetry is required by the ZIL method.

blank. When the following string compares low to the string "␣␣␣␣␣␣␣␣", then each blank is translated as itself, and no compression occurs. Thus, we can compress strings of blanks up to 256 when using asymmetric entries, while only compressing strings of blanks up to 128 when using symmetric entries. Which technique is best depends on the frequencies with which the entries occur.

4.2.2 The Tries of ALM and ZIL

Let us examine the trie that ALM might use to parse the example of Table 4, and compare it with the closest trie permitted by ZIL. Each leaf of the trie requires an encoding. ZIL requires at each internal node of the trie that a "zilch" character appear both before and after the path by which the trie is being extended with blanks. The effect of this is to give "before" and "after" encodings of each prefix. With ALM, the one "zilch" extension that is less than blank occurs after a single blank and provides an entry that encodes all strings of blanks less than length eight that compare low to a string of eight blanks. This extension encodes such strings one blank at a time in this example.

As we have seen in comparing ALM with Huffman and Hu-Tucker methods, ALM uses the same parsing strategy to decode as it does to encode. Thus, it provides very flexible encodings as well. The encodings need not have the prefix property. This is not discussed with the ZIL method [9].

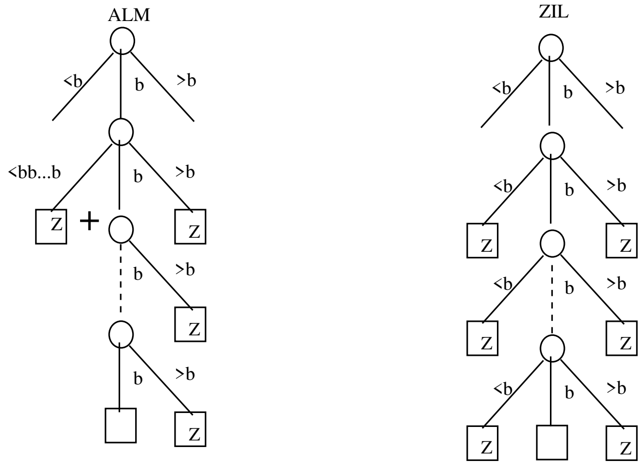


Figure 3: ALM parse trie for the dictionary of Table 4 compared with the ZIL parse trie. Blank is denoted by “b”and ”zilch” by “z”. Note thatt ZIL needs symmetric prefixes. Hence, each ZIL prefix needs two encodings. With ALM, the ”zilch” marked with a “+” encodes all blank strings shorter than the maximum length that are followed by strings that compare low to blanks.

5 Discussion

5.1 ALM Framework

ALM encoding is a framework for performing variable length order preserving translations. It gives you a new forward context method for uniquely tokenizing strings. That is, the strings considered for encoding might not occur in arbitrary contexts, but only with a constrained set of following strings. We can use ALM’s more powerful tokenization to achieve great flexibility in choosing which strings to encode. We can also use it to expand the choice of encodings so as to achieve better compression.

The ALM framework provides constraints that can be tested to ensure that the strings chosen for encoding and the encodings chosen for them are appropriate for ALM. ALM also provides the procedure which, when provided with the tables of entries and encodings, does the translation.

The Hu-Tucker method gives you a procedure for generating the encoding, i.e., you provide the alphabet, you provide the frequencies, and it generates the optimal weighted tree. The tree addresses of the leaves are a set of bit string encodings that have the prefix property and that preserve order. ALM does not provide this. Rather, it gives you a way of doing context dependent parsing so that the translatable substrings can be prefixes of other entries. What entries you choose to translate and what to translate them into remain your responsibility. So long as the context dependent order is observed for both original and encoded forms, the translation will preserve order. (One can also produce translation/compression that does not preserve order. That is ALM can do the translation for Huffman encoding as well for Hu-Tucker.)

5.2 Results

We have used ALM to solve the multifield comparison problem. It very efficiently compresses out blanks, providing a more effective solution than its predecessors. The maximum compression factors achieved for blank strings are in the range of 128:2. The multifield mapping with blank compression is implemented in the newest release of DEC Rdb. This compression is fully conformant with the SQL standard.

We have also illustrated how ALM can out-perform the Hu-Tucker "optimal" tree method. The length of the ALM encoding in our example is only about 75 per cent of the length produced by Hu-Tucker. So it should be clear that ALM can achieve very substantial compression. We point out also that translating entries with the prefix propeerty is simply a special case of context dependent translation. The same translation algorithm is effective. So, if one generates a Hu-Tucker encoding, one can use the ALM translation mechanism to do the translation. It is also possible to improve such an encoding using the ALM framework by exploiting encodings without the prefix property.

5.3 Future Work

The largest loose end is discovering methods for generating optimal encodings given the ALM framework. One can view ALM as producing a tree in which interior nodes can denote encoded entries, not just leaf nodes. How best to minimize the weight of entries in that tree is an open problem. How to minimize the weight when the entries can only appear in some contexts is a further elaboration of this problem.

An open question here is whether the order preserving compression is sufficiently good that the compression can be considered a form of order-preserving hashing. That is, are the compressed forms even approximately uniformly distributed? With perfect compression, the answer is yes, as perfect compression is only achieved when each bit of an encoding is equally likely to be either a zero or a one. How close we come to perfect compression using ALM? And is this good enough so that a "hashing" method will have search performance that is independent of the number of entries to be searched.

References

[1] Bayer, R. and McCreight, E. Organization and maintenance of large ordered indices. *Acta Informatica* 1,3(1972), 173-189.

[2] Bell, T.C., Cleary, J.G., and Witten, I.H. *Text Compression.* Prentice Hall (1990) London, UK

[3] Blasgen, M., Casey, R., and Eswaran, K. An Encoding Method for Multifield Sorting and Indexing. IBM Research Report RJ 1753 (March, 1976), Almaden Research Center, San Jose, CA.

[4] ISO (International Organization for Standardization), *ISO/IEC 9075:1992, Information Technology- Database Language SQL* July, 1992.

[5] Knuth, D. *The Art of Computing Programming. vol. 1 Fundamental Algorithms, vol.3 Sorting and Searching.* Addison Wesley (1973) Reading, MA

[6] Lorin, H. *Sorting and Sort Systems* Addison Wesley (1975) Reading, MA

[7] Moffat, A. and Zobel, J. Coding for Compression in Full-text Retrieval Systems. Data Compression Conference (1992) Snowbird, UT, 72-81.

[8] Nyberg, C., Barclay, T., Cvetampvoc. Z., Gray, J., and Lomet, D. AlphaSort: a RISC Machine Sort SFSC Technical Report 93.2 (April 1993)

[9] Zandi, A., Iyer, B. and Langdon, G. Sort Order Preserving Data compression for Extended Alphabets. Data Compression Conference (1993) Snowbird, UT, 330-339.

[10] Ziv, J., Lempel, A. Compression of individual sequences via variable-rate coding. *IEEE Trans.Information Theory* IT-24, 5 (Sept. 1978) 530-536.

Contents