# Compression of Concordances in Full-Text Retrieval Systems

**Yaacov Choueka**[1], **Aviezri S. Fraenkel**[2], **Shmuel T. Klein**[3]

[1] Department of Mathematics and Computer Science, Bar-Ilan University, Ramat Gan, Israel
[2] Dept. of Appl. Math. and Computer Science, Weizmann Institute of Science, Rehovot, Israel
[3] Graduate Library School and Computer Science Department, University of Chicago, IL 60637

The work of the third author was partially supported by a fellowship of the Ameritech Foundation

July 1988

## ABSTRACT

The *concordance* of a full-text information retrieval system contains for every different word $W$ of the database, a list $L(W)$ of "coordinates", each of which describes the exact location of an occurrence of $W$ in the text. The concordance should be compressed, not only for the savings in storage space, but also in order to reduce the number of I/O operations, since the file is usually kept in secondary memory. Several methods are presented, which efficiently compress concordances of large full-text retrieval systems. The methods were tested on the concordance of the Responsa Retrieval Project and yield savings of up to 49% relative to the non-compressed file; this is a relative improvement of about 27% over the currently used prefix-omission compression technique.

# 1. Motivation and Introduction

It is often argued that data compression techniques will in the near future become superfluous, because the price of storage media is rapidly dropping. However, a large number of works on compression were published only recently. This can be explained by the fact that while storage cost decreases, there is, at the same time, an increase in the amount of data one wishes to have in computer accessible form. Moreover, a file should be compressed not only just to save space, but often also to gain speed: the time spent on decompressing is generally largely compensated for by the savings in the number of I/O-accesses, which are typically slow, since more information can be transferred in a single read/write operation. On the other hand, compression is also useful to reduce the large amount of data which is transmitted electronically, as communication costs are still rather high.

In full-text information retrieval systems, very large files of various kinds have to be accessed frequently. First and foremost, the core of any system is its text-file, consisting generally of tens of millions of words in some natural language. Examples include the Trésor de la Langue Française (TLF) with a French database of about 150 million words, covering the literature, history and science of France from the eighteenth century to today; and the Responsa Retrieval Project (RRP) with about 60 million words of Rabbinical literature spanning more than ten centuries, written mainly in Hebrew and Aramaic. For details on RRP, the reader is referred to [4] and [8], and on full-text databases in general to [18]. It is thus natural that most of the research focused on text-compression.

One popular compression method is textual substitution. Storer & Szymanski [17] define general macro models describing such techniques: in the *external* macro model, a dictionary $D$ is given, typically containing some common words or word-fragments, and a string is compressed by replacing some of its sub-strings by (shorter) pointers to $D$. For example, Fraenkel, Mor & Perl [11] consider the case where $D$ is restricted to contain only prefixes and suffixes of words in the text. In the *internal* macro model, a string $S$ is compressed by replacing some sub-strings $T$ by pointers to previous occurrences of $T$ in $S$. Examples are the Ziv & Lempel method [24] and its variants (Welch [22], Jakobsson [13]), which are adaptive and thus need only one pass over the text. They dynamically construct a dictionary of sub-strings, which is accessed using some hashing strategy. Recently, Bentley, Sleator, Tarjan & Wei [2] proposed to replace a word $W$ by a pointer to a previous occurrence of $W$ and to maintain the list of pointers using a move-to-front policy for the most recently accessed pointer, thus yielding a locally adaptive scheme. In all the above methods, variable-length parts of the text are replaced by fixed or variable-length encodings.

On the other hand, there are *statistical* encoding techniques, which typically replace fixed-size blocks of the text (single characters, bigrams, etc.) by variable-length codewords, whose lengths are inversely related to the frequencies of the blocks in the text. These methods include among others predictive encoding (Teuhola

& Raita [20]), arithmetic coding (see for example Rissanen & Langdon [15]) and Huffman coding [12]. To obtain optimal compression, two passes over the data are needed: the first for recording the frequencies and the second for the actual encoding, but adaptive one-pass methods exist (Witten, Neal & Cleary [23] for arithmetic coding, Cormack & Horspool [7], Knuth [14] and Vitter [21] for Huffman coding) and they are shown to approach optimal efficiency in the long run. For a recent book on data compression see Storer [16].

However, most large information retrieval systems are based on inverted files. In this approach, processing of queries does not involve directly the original text files (in which key words are located using some pattern matching technique), but rather the auxiliary *dictionary* and *concordance* files. The dictionary is the list of all the different words appearing in the text and is usually ordered alphabetically. Every occurrence of every word in the database can be uniquely characterized by a sequence of numbers that give its exact position in the text. Typically, such a sequence would consist of the document number $d$, the paragraph number $p$ (in the document), the sentence number $s$ (in the paragraph) and the word number $w$ (in the sentence). The quadruple $(d, p, s, w)$ is the *coordinate* of the occurrence, and the corresponding fields will be called for short d-field, p-field, s-field and w-field. In the sequel, we assume for the ease of discussion that coordinates of every retrieval system are of this form; however, all the methods can also be applied to systems with different coordinate structure, such as book-page-line-word, etc. The concordance contains, for every word of the dictionary, the lexicographically ordered list of all its coordinates in the text; it is accessed via the dictionary that contains for every word a pointer to the corresponding list in the concordance.

Other large files which are used in some systems to enhance the retrieval process include: bitmaps, where each map is acting as an occurrence map for a given word $W$, yielding partial information on where the coordinates of $W$ are located, see [6]; permuted dictionaries, used for the processing of truncated terms, see [3]; and various grammatical files, see [1]. Examples of algorithms for the compression of large dictionaries can be found in [10], for sparse bit-vectors in [19], [5] and [9], and for permuted dictionaries in [3]. In the present work, we concentrate on the compression of the concordance, the size of which is comparable to, and often exceeds, the size of the text-file.

The concordance is kept in compressed form on secondary storage and parts of it are fetched when needed and decompressed. The compressed file is partitioned into equi-sized blocks such that one block can be read by a single I/O operation. Compression of concordances is particularly important in the context of optical storage where disk access is rather slow; also concordance files are usually huge — typically a few hundred MB, so that an additional savings of 20% can be indeed very significant. A CD-ROM has a specific amount of storage and if the whole files of the system do not fit on one disk (550 MB), then the database would have to be split between two disks (even if the surplus is very small), which is very incovenient for optical storage (there is a need to physically change disks).

In the next section we review the well-known *prefix-omission method* (POM), which is often used for the compression of dictionaries, but can easily be adapted also to concordances. In Section 3 we present some new techniques, some of which include and extend POM. In Section 4, all the methods are tested on the concordance of RRP, which, in 1981 when the statistics for the work reported herein were collected, consisted of about 37,500 documents, containing some 38 million words of running text. We are currently working on applying the new methods to the concordance of TLF, which is not simple because there exists very little structural information.

## 2. The Prefix-Omission Technique

The method is based on the observation that consecutive entries in a dictionary mostly share some leading letters. Let $x$ and $y$ be consecutive dictionary entries and let $m$ be the length (number of letters) of their longest common prefix. Then it suffices to store this common prefix only once (with $x$) and to omit it from the following entry, where instead the length $m$ will be kept. This is easily generalized to a longer list of dictionary entries, as in the following example:

| *dictionary entry* | *prefix length* | *stored suffix* |
|---|---|---|
| `FORM` | 0 | `FORM` |
| `FORMALLY` | 4 | `ALLY` |
| `FORMAT` | 5 | `T` |
| `FORMATION` | 6 | `ION` |
| `FORMULATE` | 4 | `ULATE` |
| `FORMULATING` | 8 | `ING` |
| `FORTY` | 3 | `TY` |
| `FORTHWITH` | 4 | `HWITH` |

Note that the value given for the prefix length does not refer to the string which was actually stored, but rather to the corresponding full-length dictionary entry. The compression and decompression algorithms are immediate.

If the dictionary entries are coded in standard format, with one byte per character, one could use the first byte of each entry in the compressed dictionary to store the value of $m$. There will mostly be a considerable gain, since the average length of common prefixes of consecutive entries in large dictionaries is generally much larger than 1. Even when the entries are already compressed, for example by a character by character Huffman code, one would still achieve some savings. For convenience, one could choose a fixed integer parameter $k$ and reserve the first $k$ bits of every entry to represent values of $m$ for $0 \le m < 2^k$, where $k$ is not necessarily large enough to accommodate the longest omitted prefix. In the above example, $k$ could for example be chosen as 3, and the entry corresponding to `FORMULATING` would then be $(7, \texttt{TING})$. A formal definition of POM can be found in [3], where it is called *front-end compression*. An application of the method to the compression of sparse bit-strings is described in [5].

Since the list of coordinates of any given word is ordered, adjacent coordinates will often have the same d-field, or even the same d- and p-fields, and sometimes, especially for high frequency words, identical d-, p- and s-fields. Thus POM can be adapted to the compression of concordances, where to each coordinate a *header* is adjoined, giving the *number of fields* which can be copied from the preceding coordinate; these fields are then omitted. For instance in our model with coordinates $(d, p, s, w)$, it would suffice to keep a header of 2 bits. The four possibilities are: don't copy any field from the previous coordinate, copy the d-field, copy d- and p-field and copy d-, p- and s-field. Obviously, different coordinates cannot have all four fields identical.

For convenient computer manipulation, one generally chooses a fixed length for each field, which therefore has to be large enough to represent the maximal possible values. However, most stored values are small, thus there is usually much wasted space in each coordinate. In some situations, some space can be saved at the expense of a longer processing time, as in the following example.

At RRP, the maximal length of a sentence is 676 words! Such long sentences can be explained by the fact that in the Responsa literature punctuation marks are often omitted or used very scarcely. At TLF, there is even a "sentence" of more than 2000 words (a modern poem). Since on the other hand most sentences are short and it was preferred to use only field-sizes which are multiples of half-bytes, the following method is used: the size of the w-field is chosen to be one byte (8 bits); any sentence of length $\ell > 256$ words, such that $\ell = 80k + r$ $(0 \leq r < 80)$, is split into $k$ units of 80 words, followed (if $r > 0$) by a sentence of $r$ words. These sentences form only a negligible percentage of the database. While resolving the storage problem, the insertion of such "virtual points" in the middle of a sentence creates some problems for the retrieval process. When in a query one asks to retrieve occurrences of keywords $A$ and $B$ such that $A$ and $B$ are adjacent or that no more than some small number of words appear between them, one usually does not allow $A$ and $B$ to appear in different sentences. This is justified, since "adjacency" and "near vicinity" operators are generally used to retrieve expressions, and not the coincidental juxtaposition of $A$ at the end of a sentence with $B$ at the beginning of the following one. However in the presence of virtual points, the search should be extented also into neighbouring "sentences", if necessary, since the virtual points are only artificial boundaries which might have split some interesting expression. Hence this solution further complicates the retrieval algorithms.

The methods presented in the next section not only yield improved compression, but also get rid of the virtual points.

## 3. Using Variable-Length Fields

The basic idea of all the new methods is to allow the p-, s- and w-fields to have variable length. As in POM, each compressed coordinate will be prefixed by a header which will encode the information necessary to decompress the coordinate. The methods differ in their interpretation of the header. The choice of the length of every field is based on statistics gathered from the entire database on the distribution of the values in each field. Thus for dynamically changing databases, the compression method would need frequent updates, so that the methods are more suitable for retrieval systems with static databases. However, if the text changes only slowly, say it is a large corpus to which from time to time some documents are adjoined which have characteristics similar to the documents already in the corpus, then the methods will still perform well, though not optimally.

The codes in the header can have various interpretations: they can stand for a length $\ell$, indicating that the corresponding field is encoded in $\ell$ bits; they can stand for a certain value $v$, indicating that the corresponding field contains that value; they can finally indicate that no value for the corresponding field is stored and that the value of the preceding coordinate should be used. This is more general than the prefix-omission technique, since one can decide for every field individually whether or not to omit it, while in POM, the p-field is only omitted if the d-field is, etc.

The d-field is treated somewhat differently. Since this is the highest level of the hierarchy in our model, this field may contain also very large numbers (there are rarely 500 words in a sentence or 500 sentences in a paragraph, but a corpus may contain tens of thousands of documents). Moreover, the d-fields of most coordinates will contain values, in the representation of which one can save at most one or two bits, if at all. On the other hand, the d-field is the one where the greatest savings are achieved by POM. Thus we shall assume in the sequel that for the d-field, we just keep one bit in the header, indicating whether the value of the preceding coordinate should be copied or not; if not, the d-field will appear in its entire length.

We now describe the specific methods in detail.

**A.** The simple method. The header contains codes for the size (in bits) of every field.

**(i)** Allocate two bits for each of the p-, s- and w-fields, giving four possible choices for each.

We consider the following variations:

**a.** One of the possible codes indicates the omission of the field, thus we are left with only 3 possible choices for the length of each field.

**b.** The four choices are used to encode field-lengths, thus not allowing the use of the preceding coordinate.

**c.** Use **a** for the p- and s-fields, and **b** for the w-field.

Method **A(i)c** is justified by the fact that consecutive coordinates having the same value in their w-field are rare (3.5% of the concordance at RRP). The reason is that this corresponds to a certain word appearing in the same relative location in different sentences, which is mostly a pure coincidence; on the other hand consecutive coordinates having the same value in one of their other fields correspond to a certain word appearing more than once in the same sentence, paragraph or document, and this occurs frequently. For instance, at RRP, 23.4% of the coordinates have the same s-field as their predecessors, 41.7% have the same p-field and 51.6% have the same d-field.

Note that the header does not contain the binary encoding of the lengths, since this would require a larger number of bits. By storing a *code* for the lengths the header is kept smaller, but at the expense of increasing decompression time, since a table is needed which translates the codes into actual lengths. This remark applies also to the subsequent methods.

**(ii)** Allocate three bits in the header for each of the p-, s- and w-fields, giving 8 possible choices for each.

The idea of **(ii)** is that by increasing the number of possibilities (and hence the overhead for each coordinate), the range of possible values can be partitioned more efficiently, which should lead to savings in the remaining part of the coordinate. Again three methods corresponding to **a**, **b** and **c** of **(i)** were checked.

**B.** Using some fields to encode frequent values.

For some very frequent values, the code in the header will be interpreted directly as one of the values, and not as the length of the field in which they are stored. Thus the corresponding field can be omitted in all these cases. However, the savings for the frequent values come at the expense of reducing the number of possible choices for the lengths of the fields for the less frequent values. For instance, at RRP, the value 1 appears in the s-field of more than 9 million coordinates (about 24% of the concordance), thus all these coordinates will have no s-field in their compressed form, and the code in the part of the header corresponding to the s-field will be interpreted as "value 1 in the s-field".

**(i)** Allocate 2 bits in the header for each of the p-, s- and w-fields; one of the codes points to the most frequent value.

**(ii)** Allocate 3 bits in the header for each of the p-, s- and w-fields; three of the codes point to the 3 most frequent values.

There is no subdivision into methods **a**, **b** and **c** as in **A** (in fact the method used corresponds to **a**), because we concluded from our experiments that it is worth to keep the possibility of using the previous coordinate in case of equal values in some field. Hence one code was allocated for this purpose, which left only 2 codes to encode the field-lengths in **(i)** and 4 codes in **(ii)**. For **(ii)** we experimented also with allowing 2 or 4 of the 8 possible choices to encode the 2 or 4 most frequent values; however, on our data, the optimum was always obtained for 3. There is some redundancy in the case of consecutive coordinates having both the same value in some field, and this value being the most frequent one. There are then two possibilities to encode the second coordinate using the same number of bits. In such a case, the code for the frequent value should be preferred over the one pointing to the previous coordinate, as decoding of the former is usually faster.

**C.** Combining methods **A** and **B.**

Choose individually for each of the p-, s- and w-fields, the best of the previous methods.

**D.** Encoding length-combinations.

If we want to push the idea of **A** further, we should have a code for *every* possible length of a field, but the maxima of the values can be large. For example, at RRP, one needs 10 bits for the maximal value of the w-field, 9 bits for the s-field and 10 bits for the p-field. This would imply a header length of 4 bits for each of these fields, which cannot be justified by the negligible improvement over method **A(ii)**.

The size of the header can be reduced by replacing the three codes for the sizes of the p-, s- and w-fields by a single code in the following way. Denote by $l_p$, $l_s$ and $l_w$ the lengths of the p-, s- and w-fields respectively, i.e., the sizes (in bits) of the binary representations without leading zeros of the values stored in them. In our model $1 \leq l_p, l_s, l_w \leq 10$, so there are up to $10^3$ possible triplets $(l_p, l_s, l_w)$. However, most of these length-combinations occur only rarely, if at all. At RRP, the 255 most frequent $(l_p, l_s, l_w)$-triplets account already for 98.05% of the concordance. Therefore

**(i)** Allocate 9 bits as header, of which 1 bit is used for the d-field; 255 of the possible codes in the remaining 8 bits point to the 255 most frequent $(l_p, l_s, l_w)$-triplets; the last code is used to indicate that the coordinate corresponds to a "rare" triplet, in which case the p-, s- and w-fields appear already in their decompressed form.

Although the "compressed" form of the rare coordinates, including a 9-bit header, may in fact need more space than the original coordinate, we still save on the average.

Two refinements are now superimposed. We first note that one does not need to represent the integer 0 in any field. Therefore one can use a representation of the integer $n-1$ in order to encode the value $n$, so that only $\lfloor \log_2(n-1) \rfloor + 1$ bits are needed instead of $\lfloor \log_2 n \rfloor + 1$. This may seem negligible, because only one bit is saved and only when $n$ is a power of 2, thus for very few values of $n$. However, the first few of these values, 1, 2 and 4, appear very frequently, so that in fact this yields a significant improvement. At RRP, the total size of the compressed p-, s- and w-fields (using method **D**) was further reduced by 7.4%, just by shifting the stored values from $n$ to $n-1$.

The second refinement is based on the observation that since we know from the header the exact length of each field, we know the position of the left-most 1 in it, so that this 1 is also redundant. The possible values in the fields are partitioned into classes $\mathcal{C}_i$ defined by $\mathcal{C}_0 = \{0\}$, $\mathcal{C}_i = \{\ell : 2^{i-1} \le \ell < 2^i\}$, and the header gives for the values in each of the p-, s- and w-fields, the indices $i$ of the corresponding classes. Therefore if $i \le 1$, there is no need to store any additional information because $\mathcal{C}_0$ and $\mathcal{C}_1$ are singletons, and for $\ell \in \mathcal{C}_i$ for $i > 1$, only the $i-1$ bits representing the number $\ell - 2^{i-1}$ are kept. For example, suppose the values in the p-, s- and w-fields are 3, 1 and 28. Then the encoded values are 2, 0 and 27 which belong to $\mathcal{C}_2$, $\mathcal{C}_0$ and $\mathcal{C}_5$ respectively. The header thus points to the triplet $(2, 0, 5)$ (assuming that this is one of the 255 frequent ones) and the rest of the coordinate consists of the five bits 01011, which are parsed from left to right as 1 bit for the p-field, 0 bits for the s-field and 4 bits for the w-field. A similar idea was used in [9] for encoding run-lengths in the compression of sparse bit-vectors.

**(ii)** Allocate 8 bits as header of which 1 bit is used for the d-field; the remaining 7 bits are used to encode the 127 most frequent $(l_p, l_s, l_w)$-triplets.

The 127 most frequent triplets still correspond to 85.19% of the concordance at RRP. This is therefore an attempt to save one bit in the header of each coordinate at the expense of having more non-compressed coordinates.

Another possibility is to extend method **D** also to the d-field. Let $b$ be a Boolean variable corresponding to the two possibilities for the d-field, namely T = the value is identical to that of the preceding coordinate, thus omit it, or F = different value, keep it. We therefore have up to 2000 quadruples $(b, l_p, l_s, l_w)$, which are again sorted by decreasing frequency.

**(iii)** Allocate 8 bits as header; 255 of the codes point to the 255 most frequent quadruples.

At RRP, these 255 most frequent quadruples cover 87.08% of the concordance. For the last two methods, one could try to get better results by compressing also some of the coordinates with the non-frequent length combinations, instead of storing them in their decompressed form. We did not, however, pursue this possibility.

## Encoding

After choosing the appropriate compression method, the concordance is scanned sequentially and each coordinate is compressed with or without using the preceding one. For each of the above methods, the length of the header is constant, thus the set of compressed coordinates forms a prefix-code. Therefore the compressed coordinates, which have variable lengths, can simply be concatenated. The compressed concordance consists of the resulting very long bit-string. This string is partitioned into blocks of equal size, the size corresponding to the buffer-size of a read/write operation. If the last coordinate in a block does not fit there in its entirety, it is moved to the beginning of the next block. The first coordinate of each block is considered as having no predecessor, so that if in the original encoding process a coordinate which is the first in a block referred to the previous coordinate, this needs to be corrected. This allows now to access each block individually, while adding only a negligible number of bits to each block.

## Decoding

Note that for a static information retrieval system, encoding is done only once (when building the database), whereas decoding directly affects the response-time for on-line queries. In order to increase the decoding speed, we use a small precomputed table $\mathcal{T}$ which is stored in internal memory. For a method with header length $k$ bits, this table has $2^k$ entries. In entry $i$ of $\mathcal{T}$, $0 \leq i < 2^k$, we store the relevant information for the header consisting of the $k$-bit binary representation of the integer $i$.

For the methods in **A**, the relevant information simply consists of the lengths, $P$, $S$ and $W$, of the p-, s- and w-fields (recall that we assume that only one bit is kept in the header for the d-field, so either the d-field appears in its entire length $D$, which is constant, or it is omitted), and of the sum of all these lengths (including $D$), which is the length of the remaining part of the coordinate. We shall use the following notations: for a given internal structure of a decompressed coordinate, let $h_d$, $h_p$, $h_s$ and $h_w$ be the indices of the leftmost bit of the d-, p-, s- and w-fields respectively, the index of the rightmost bit of a coordinate being 0. For example with a 4 byte coordinate and one byte for each field we would have $h_d = 31$, $h_p = 23$, $h_s = 15$ and $h_w = 7$; these values are constant for the entire database. COOR and LAST are both addresses of a contiguous space in memory in which a single decompressed coordinate can fit (hence of length $h_d+1$ bits). The procedure SHIFT$(X, y, z)$ shifts the substring of $X$ which is obtained by ignoring its $y$ rightmost bits, by $z$ bits to the left. Then the following loop could be used for the decooding of a coordinate:

| | | |
|---|---|---|
| 1. | **loop** while there is more input or until a certain coordinate is found | |
| 2. | $H \leftarrow$ next $k$ bits | ⟦ read header ⟧ |
| 3. | $(\mathrm{TOT}, P, S, W) \leftarrow \mathcal{T}(H)$ | ⟦ decode header using table ⟧ |
| 4. | COOR $\leftarrow$ next TOT bits | ⟦ right justified suffix of coordinate ⟧ |

5. $\quad$ SHIFT(COOR, $W, h_w - W$) $\quad$ ⟦ move d-, p- and s-field ⟧
6. $\quad$ SHIFT(COOR, $h_w + S, h_s - S$) $\quad$ ⟦ move d- and p-field ⟧
7. $\quad$ SHIFT(COOR, $h_s + P, h_p - P$) $\quad$ ⟦ move d-field ⟧
8. $\quad$ **if** TOT $= P + S + W$ **then** copy d-field from LAST
9. $\quad$ **if** $P = 0$ **then** copy p-field from LAST
10. $\quad$ **if** $S = 0$ **then** copy s-field from LAST
11. $\quad$ **if** $W = 0$ **then** copy w-field from LAST
12. $\quad$ LAST $\leftarrow$ COOR
13. **end** of loop

There is no need to initialize LAST, since the first coordinate of a block never refers to the preceding coordinate.

For the methods in **B** and **C**, we store sometimes actual values, and not just the lengths of the fields. This can be implemented by using negative values in the table $\mathcal{T}$. For example, if $P = -2$, this could be interpreted as "value 2 in the p-field". Note that when the value stored in a field is given by the header, this field has length 0 in the remaining part of the coordinate. Thus we need the following updates to the above algorithm: line 3 is replaced by

$(\mathrm{TOT}, P1, S1, W1) \leftarrow \mathcal{T}(H)$
**if** $P1 < 0$ **then** $P \leftarrow 0$ **else** $P \leftarrow P1$

and statements similar to the latter for the s- and w-fields. After statement 11 we should insert

**if** $P1 < 0$ **then** put $-P1$ in p-field of COOR

and similar statements for the s- and w-fields.

The decoding of the methods in **D** is equivalent to that of **A**. The only difference is in the preparation of the table $\mathcal{T}$ (which is done only once). While for **A** to each field correspond certain fixed bits of the header which determine the length of that field, for **D** the header is non-divisible and represents the lengths of all the fields together. This does not affect the decoding process, since in both methods a table-lookup is used to interpret the header. An example of the encoding and decoding processes appears in the next section.

# 4. Experimental results

All the methods of the previous section were compared on the concordance of RRP. We are actually applying them also to TLF, but all the figures below were extracted from the RRP database. Each coordinate had a $(d, p, s, w)$-structure and was of length 6 bytes (48 bits). Using POM, the average length of a compressed coordinate was 4.196 bytes, i.e., a compression gain of 30%.

In order to apply the new methods, the following statistics were collected in a sequential scan of the concordance.

1. For integers $i$, $1 \leq i < 2^{10}$, how often does $i$ appear in each of the p-, s- and w-fields (for **A(i)b**, **A(ii)b**, **B** and **C**).
2. Same as in 1, but where an appearance of $i$ in a ceratin field was not counted when the previous coordinate had the same value $i$ in the same field (for **A(i)a** and **A(ii)a**).
3. The frequencies of the possible combinations of field-lengths (for **D(i)** and **D(ii)**), and the frequencies of the possible quadruplets $(b, l_p, l_s, l_w)$ (for **D(iii)**).

Table 1 gives the frequencies of the first few values in each of the p-, s- and w-fields, both with and without taking into account the previous coordinate. The frequencies are given in cummulative percentages, e.g., the row entitled s-field contains in the column headed $i$ the percentage of coordinates having a value $\leq i$ in their s-field. We have also added the values for which the cummulative percentage first exceeds 99%.

**Table 1:** *Distribution of values stored in p-, s- and w-fields*

| Value | | 1 | 2 | 3 | 4 | 5 | 79 | 83 | 87 | 93 | 119 | 120 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ignoring | p-field | 14.1 | 35.2 | 46.5 | 54.2 | 60.2 | | 99 | | | | |
| preceding | s-field | 24.2 | 40.2 | 51.1 | 58.8 | 64.5 | 99 | | | | | |
| coordinate | w-field | 3.0 | 5.8 | 8.6 | 11.4 | 14.0 | | | | | 99 | |
| using | p-field | 9.6 | 25.2 | 36.5 | 45.0 | 51.7 | | | | 99 | | |
| preceding | s-field | 17.9 | 33.0 | 44.3 | 52.6 | 58.9 | | | 99 | | | |
| coordinate | w-field | 1.9 | 4.4 | 7.1 | 9.7 | 12.4 | | | | | | 99 |

As one can see, the first four values in the p- and s-fields account already for half of the concordance. This means that most of the paragraphs consist of only a few sentences and most of the documents consist of only a few paragraphs. The figures for the w-field are different, because short sentences are not preponderant. While the (non-cummulative) frequency of the values $i$ in the s-field is a clearly decreasing function of $i$, it is interesting to note the peek at value 2 for the p-field. This can be explained by the specific nature of the Responsa literature, in which most of the documents have a question-answer structure. Therefore the first paragraph of a document usually contains just a short question, whereas the answer, starting from the second paragraph, may be much longer.

When all the coordinates are considered (upper half of Table 1), the percentages are higher than the corresponding percentages for the case where identical fields in adjacent coordinates are omitted (lower half of Table 1). This means that the idea of copying certain fields from the preceding coordinate yields to savings which are, for the small values, larger than could have been expected from knowing their distribution in the non-compressed concordance.

Using the information we collected from the concordance, we have checked all the possible variants for each of the methods in **A** and **B**. Since the maximal field-length (10 for the p- and w-fields) had to be included for every variant, the number of variants to be checked for the p- and w-fields was only $\binom{9}{2}$ for **A(i)a**, $\binom{9}{3}$ for **A(i)b**, $\binom{9}{6}$ for **A(ii)a**, $\binom{9}{7}$ for **A(ii)b**, $\binom{9}{1}$ for **B(i)** and $\binom{9}{3}$ for **B(ii)**; the maximum for the s-field is 9, so the corresponding numbers for the s-field are obtained from the above, by substituting 8 for 9 in the binomial coefficients. Table 2 lists for each of the methods the variant for which maximal compresion was achieved. The numbers in boldface are the frequent values which are used in methods **B** and **C**, the other numbers refer to the lengths of the fields. The value 0 indicates that the field of the preceding coordinate should be copied.

Table 2: *Optimal variants of the methods*

| Method | p-field | s-field | w-field |
|---|---|---|---|
| **A(i)a** | 0 2 5 10 | 0 2 5 9 | 0 4 6 10 |
| **A(i)b** | 1 3 5 10 | 1 3 5 9 | 3 5 6 10 |
| **A(ii)a** | 0 1 2 3 4 5 6 10 | 0 1 2 3 4 5 6 9 | 0 1 3 4 5 6 7 10 |
| **A(ii)b** | 1 2 3 4 5 6 7 10 | 1 2 3 4 5 6 7 9 | 1 2 3 4 5 6 7 10 |
| **B(i)** | 0 **2** 4 10 | 0 **1** 4 9 | 0 **4** 6 10 |
| **B(ii)** | 0 **1 2 3** 3 4 5 10 | 0 **1 2 3** 3 4 5 9 | 0 **3 4 5** 3 5 6 10 |
| **C** | 0 2 5 10 | 0 **1 2 3** 3 4 5 9 | 3 5 6 10 |

The optimal variants for the methods **A(ii)** are not surprising: since most of the stored values are small, one could expect the optimal partition to give priority to small field-lengths. For method **C**, each field is compressed by the best of the other methods, which are **A(i)a** for the p-field, **B(ii)** for the s-field and **A(i)b** for the w-field, thus requiring a header of $1+2+3+2=8$ bits (including one bit for the d-field).

The entries of Table 2 were computed using the first refinement mentioned in the description of method **D**, namely storing $n-1$ instead of $n$. The second refinement (dropping the leftmost 1) could not be applied, because it is not true that the leftmost bit in every field is a 1. Thus for all the calculations with methods **A** and **B**, an integer $n$ was supposed to require $\lfloor \log_2(n-1) \rfloor + 1$ bits for $n > 1$ and one bit for $n = 1$.

As an example for the encoding and decoding processes, consider method **C**, and a coordinate structure with $(h_d, h_p, h_s, h_w) = (8, 8, 8, 8)$, i.e., one byte for each field. The coordinate we wish to process is $(159, 2, 2, 35)$. Suppose further that only the value in the d-field is the same as in the previous coordinate. Then the length $D$ of the d-field is 0; in the p-field the value 1 is stored, using two bits; nothing is stored in the s-field, because 2 is one of the frequent values and directly referenced by the header; in the w-field the value 34 is stored, using 6 bits. The possibble options for the header are numbered from left to right as they appear in Table 2, hence the header of this coordinate is 0-10-011-11, where dashes separating the parts corresponding to different fields have been added for clarity; the remaining part of the coordinate is 01-100010. The table $\mathcal{T}$ has $2^8 = 256$ entries; at entry 79 (= 01001111 in binary) the values stored are $(\mathrm{TOT}, P1, S1, W1) = (8, 2, -2, 6)$. When decoding the compressed coordinate 0100111101100010, the leftmost 8 bits are considered as header and converted to the integer 79. Table $\mathcal{T}$ is then accessed with that index, retrieving the 4-tuple $(8, 2, -2, 6)$ which yields the values $(P, S, W) = (2, 0, 6)$. The next TOT = 8 bits are therefore loaded into COOR of size 4 bytes, and after the three shifts we get

$$\mathrm{COOR} = 00000000 - 00000010 - 00000000 - 00100010.$$

Since $\mathrm{TOT} = P + S + W$ the value of the d-field is copied from the last coordinate. Since $P1 < 0$, the value $-S1 = 2$ is put into the s-field.

In Table 3, the most frequent length combinations for the methods in **D** are given, together with their relative weights (in %). For **D(i)** the triplets $(l_p, l_s, l_w)$ are listed and for **D(ii)** the quadruples $(b, l_p, l_s, l_w)$, $b$ being the Boolean variable corresponding to the d-field. In order to evaluate the influence of the first refinement, statistics were also collected using a $\lfloor \log_2 n \rfloor$ bit representation for an integer $n$ (thus again omitting the leftmost 1).

**Table 3:** *Most frequent length combinations*

| using $\lfloor \log_2 n \rfloor$ bits | | | | **D(i)** | | | | **D(ii)** | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $l_p$ | $l_s$ | $l_w$ | % | $l_p$ | $l_s$ | $l_w$ | % | $b$ | $l_p$ | $l_s$ | $l_w$ | % |
| 1 | 1 | 5 | 2.31 | 0 | 0 | 5 | 3.18 | T | 0 | 0 | 5 | 2.33 |
| 1 | 2 | 5 | 2.16 | 0 | 0 | 4 | 2.51 | T | 0 | 0 | 4 | 1.99 |
| 2 | 1 | 5 | 1.92 | 2 | 0 | 5 | 2.50 | T | 0 | 0 | 3 | 1.46 |
| 2 | 0 | 5 | 1.90 | 1 | 0 | 5 | 2.25 | F | 2 | 0 | 5 | 1.33 |
| 1 | 0 | 5 | 1.84 | 3 | 0 | 5 | 2.09 | T | 1 | 0 | 5 | 1.30 |
| 1 | 2 | 4 | 1.82 | 2 | 0 | 4 | 1.83 | F | 3 | 0 | 5 | 1.28 |

Shifting the stored values from $n$ to $n-1$ completely changed the order of the list of length-combinations, which was ordered by decreasing frequency. For example the two most frequent triplets for **D(i)**, $(0, 0, 5)$ and $(0, 0, 4)$, appear in the list of the method without the first refinement (leftmost column of Table 3) only in positions 42 and 50 respectively.

Table 4 summarizes the results. For each method the average length in bytes of the compressed coordinate (using the best variant) is given. Compression is in percent and defined as

$$\text{Compression} = \left(1 - \frac{\text{average size of compressed coordinate}}{\text{size of non-compressed coordinate}}\right) \times 100.$$

The improvement over POM is defined as the relative additional savings (in percent) when substituting the new methods for POM.

On our data, the best method was **D(i)**, followed by **C** which clearly improves both **A** and **B**. Nevertheless, the results depend heavily on the statistics of the specific system at hand, so that for another database, other methods could be preferable.

**Table 4:** *Compression results*

| Method | | | length of the header in bit | average coordinate length (in byte) | compression % | improvement over POM % |
|---|---|---|---|---|---|---|
| POM | | | 2 | 4.196 | 30.1 | – |
| **A** | **(i)** | **a** | 7 | 3.166 | 47.2 | 24.6 |
| | | **b** | 7 | 3.259 | 45.7 | 22.3 |
| | | **c** | 7 | 3.151 | 47.5 | 24.9 |
| | **(ii)** | **a** | 10 | 3.281 | 45.3 | 21.8 |
| | | **b** | 10 | 3.444 | 42.6 | 17.9 |
| | | **c** | 10 | 3.292 | 45.1 | 21.5 |
| **B** | **(i)** | | 7 | 3.327 | 44.6 | 20.7 |
| | **(ii)** | | 10 | 3.279 | 45.4 | 21.9 |
| **C** | | | 8 | 3.144 | 47.6 | 25.1 |
| **D** | **(i)** | | 9 | 3.082 | 48.6 | 26.6 |
| | **(ii)** | | 8 | 3.591 | 40.2 | 14.4 |
| | **(iii)** | | 8 | 3.497 | 41.7 | 16.7 |

# 5. Conclusion

The new methods efficiently compress the concordance, which is the largest and most frequently accessed file of a full-text retrieval system. They achieve a significant improvement over the known prefix omission technique, while still being easy to implement and allowing for fast decompression. The main target of the efforts was to try to eliminate or at least reduce the unused space in the coordinates.

Note that this can easily be achieved by considering the entire database as a single long run of words, which we could index sequentially from 1 to $N$, $N$ being the total number of words in the text. Thus $\lfloor \log_2 N \rfloor + 1$ bits would be necessary per coordinate. However, the hierarchical structure is lost, so that, for example, queries asking for the co-occurrence of several words in the same sentence or paragraph are much harder to process. Moreover, when a coordinate is represented by a single, usually large, number, we lose also the possibility to omit certain fields which could be copied from preceding coordinates. A hierarchical structure of a coordinate is therefore preferable for the retrieval algorithms. As can be seen from Table 4, some of the new compression methods even outperform the simple method of sequentially numbering the words, since the latter would imply at the RRP database a coordinate length of 26 bits = 3.25 bytes.

## References


[1] **Attar R., Choueka Y., Dershowitz N., Fraenkel A.S.,** KEDMA — Linguistic tools for retrieval systems, *J. ACM* **25** (1978) 52–66.

[2] **Bentley J.L., Sleator D.D., Tarjan R.E., Wei V.K.,** A locally adaptive data compression scheme, *Comm. ACM* **29** (1986) 320–330.

[3] **Bratley P., Choueka Y.,** Processing truncated terms in document retrieval systems, *Inf. Processing & Management* **18** (1982) 257–266.

[4] **Choueka Y.,** Full text systems and research in the humanities, *Computers and the Humanities* **XIV** (1980) 153–169.

[5] **Choueka Y., Fraenkel A.S., Klein S.T., Segal E.**, Improved hierarchical bit-vector compression in document retrieval systems, *Proc. 9-th ACM-SIGIR Conf.,* Pisa; ACM, Baltimore, MD (1986) 88–96.

[6] **Choueka Y., Fraenkel A.S., Klein S.T., Segal E.**, Improved Techniques for Processing Queries in Full-Text Systems, *Proc. 10-th ACM-SIGIR Conf.,* New Orleans (1987) 306–315.

[7] **Cormack G.V., Horspool R.N.,** Algorithms for adaptive Huffman codes, *Inf. Proc. Letters* **18** (1984) 159–166.

[8] **Fraenkel A.S.,** All about the Responsa Retrieval Project you always wanted to know but were afraid to ask, Expanded Summary, *Jurimetrics J.* **16** (1976) 149–156.

[9] **Fraenkel A.S., Klein S.T.**, Novel compression of sparse bit-strings — preliminary report, *Combinatorial Algorithms on Words*, NATO ASI Series Vol. **F12**, Springer Verlag, Berlin (1985) 169–183.

[10] **Fraenkel A.S., Mor M.,** Combinatorial compression and partitioning of large dictionaries, *The Computer Journal* **26** (1983) 336–343.

[11] **Fraenkel A.S., Mor M., Perl Y.,** Is text compression by prefixes and suffixes practical? *Acta Informatica* **20** (1983) 371–389.

[12] **Huffman D.,** A method for the construction of minimum redundancy codes, *Proc. of the IRE* **40** (1952) 1098–1101.

[13] **Jakobsson M.,** One pass text compression with a subword dictionary, *J. Amer. Soc. for Inf. Sc.* **39** (1988) 262–269.

[14] **Knuth D.E.,** Dynamic Huffman coding, *J. of Algorithms* **6** (1985) 163–180.

[15] **Rissanen J., Langdon G.G.,** Arithmetic coding, *IBM J. Res. Dev.* **23** (1979) 149–162.

[16] **Storer J.A.,** *Data Compression, Methods and Theory*, Computer Science Press, Rockville, Maryland (1988).

[17] **Storer J.A., Szymanski T.G.,** Data compression via textual substitution, *J. ACM* **29** (1982) 928–951.

[18] **Tenopir C.,** Full text databases, *Annual Review of Inf. Sc. and Technology* **19** (1984) 215–246.

[19] **Teuhola J.,** A compression method for clustered bit-vectors, *Inf. Proc. Letters* **7** (1978) 308–311.

[20] **Teuhola J., Raita T.,** Text compression using prediction, *Proc. 9-th ACM-SIGIR Conf.,* Pisa; ACM, Baltimore, MD (1986) 97–102.

[21] **Vitter J.S.,** Design and analysis of dynamic Huffman coding, *J. ACM* **34** (1987) 825–845.

[22] **Welch T.A.,** A technique for high-performance data compression, *IEEE Computer* **17** (June 1984) 8–19.

[23] **Witten I.H., Neal R.M., Cleary J.G.,** Arithmetic coding for data compression, *Comm. ACM* **30** (1987) 520–540.

[24] **Ziv J., Lempel A.,** A universal algorithm for sequential data compression, *IEEE Trans. on Inf. Th.* **IT–23** (1977) 337–343.