Information Processing & Management Vol. 18, No. 5, pp. 257–266, 1982
Printed in Great Britain.

0306–4573/82/050257–10$03.00/0
Pergamon Press Ltd.

PROCESSING TRUNCATED TERMS IN DOCUMENT RETRIEVAL SYSTEMS

PAUL BRATLEY and YAACOV CHOUEKA†
Département d'informatique et de recherche opérationnelle, Université de Montréal, C.P. 6128, Succursale "A", Montreal P.Q., Canada H3C 3J7

(*Received for publication* 13 *March* 1982)

Abstract—In a typical inverted-file full-text document retrieval system, the user submits queries consisting of strings of characters combined by various operators. The strings are looked up in a text-dictionary which lists, for each string, all the places in the database at which it occurs. It is desirable to allow the user to include in his query truncated terms such as X^*, $^*X$, $^*X^*$, or X^*Y, where X and Y are specified strings and * is a variable-length-don't-care character, that is, * represents an arbitrary, possibly empty, string. Processing these terms involves finding the set of all words in the dictionary that match these patterns. How to do this efficiently is a long-standing open problem in this domain.

In this paper we present a uniform and efficient approach for processing all such query terms. The approach, based on a "permuted dictionary" and a corresponding set of access routines, requires essentially one disk access to obtain from the dictionary all the strings represented by a truncated term, with negligible computing time. It is thus well suited for on-line applications. Implementation is simple, and storage overhead is low: it can be made almost negligible by using some specially adapted compression techniques described in the paper.

The basic approach is easily adaptable for slight variants, such as fixed (or bounded) length don't-care characters, or more complex pattern matching templates.

1. INTRODUCTION

In this paper a solution is proposed to an often-quoted problem in inverted-file full-text document retrieval systems. The problem has to do with locating all words, in a given text T, that contain a specified substring while satisfying various "don't-care" conditions. A set of files and algorithms are described for processing these "truncated terms" that often occur in the formulation of queries.

In order to put the problem in its appropriate context, we begin in the next section by sketching the main characteristics of full-text retrieval systems with inverted files, and this is followed in Section 3 by the definition of the problem. The basic solution is given in Section 4, with some variants and refinements following in Section 5. Finally the storage issue is addressed in Section 6, where some compression techniques are described.

2. FULL-TEXT SYSTEMS

The aim of a document retrieval system is to select from a usually large set of documents those that are relevant to a specific query submitted by some user. Besides the conventional indexing approach (which was in fact the only one practical before the advent of computers), another approach, commonly known now as the *full-text method*, has attracted much attention in recent years, both on the research-experimental and on the commercial-operational levels. Conceived and described by HORTY [8] as early as 1962, the method was adopted and implemented towards the end of the sixties in a number of retrieval systems with quite large databases (mostly of legal content), among which one may cite the OBAR, DATUM, FLITE (also LITE) and RESPONSA systems, the latter having a database of historical Rabbinical

†On leave from the Department of Mathematics and Computer Science, and the Institute for Information Retrieval and Computational Linguistics (The Responsa Project), Bar-Ilan University, Ramat-Gan, Israël.

documents in Hebrew. (References on these systems and the ones mentioned below can be found in the excellent bibliography [5], which also has an index by system acronyms). Because of the technology then available, these systems were basically of a batch processing nature. In the mid-seventies, however, several on-line full-text systems were available (some of them on a commercial basis), with, characteristically, very large databases–tens of millions of words of running text, at the very least. We mention, as typical examples, the LEXIS, JURIS and QUIK-LAW (also Q/L) systems (the Responsa Project has also recently introduced an on-line version). Computer manufacturers and software houses followed the trend and soon were marketing on-line full-text retrieval packages, adaptable to different databases; one thinks, for example, of STAIRS [15] from IBM, DATA/CENTRAL [6] from DG Associates, BASIS [1] from Battelle's Columbus Laboratories, and the recent DOCU/MASTER [4] from Turnkey Systems. Because of the large amount of storage and processing needed for full-text systems, most of the above-mentioned packages and systems were designed to operate with big powerful computers. Lately however there is a growing interest in designing full-text systems that are more tailored to a minicomputer environment; this is the goal of the EUREKA [11] and DOMESTIC [9] projects, among others.

Although very markedly different in their query formulation languages, their repertoire of commands and operators, the fine details of their file structure, their display and browsing capabilities and the various retrieval tools that they make available to the user, all of these full-text systems share a common basic retrieval philisophy and essentially similar systems architecture. Three main ideas are the pivot points of this approach. First, the full documents without any *a priori* manual or automatic indexing are stored in the computer. No keywords are assigned, and almost no formatting of the documents' information is done. Second, the user submits his search topic to the system by reducing it to a set of words, expressions, phrases, citations, etc., that he assumes will occur in any document pertinent to his query. He can add distance constraints (sometimes called also positional logic or metrical operators) as well as boolean conditions (AND, OR, BUT NOT) on these expressions, asking for example for all documents containing "*differential equations* and (*group theory* or *Lie theory*) in the same paragraph", or "*curriculum* within 5 words of *information sciences* and such that *high-school* does not occur in the two neighbouring sentences", etc. Third, because of the size of the database-anywhere from millions of words to billions of characters- searching the text linearly for the query's given terms, expressions and constraints is not practical even with the fastest string-matching algorithms. Inverted files of some sort have to be built and used in the search process: hence the name *inverted-file full-text systems* sometimes assigned to these systems.

Besides the text file T mentioned above, one has also usually a *dictionary* (or *index*) file, and a *reference* (or *postings* or *concordance*) file. The dictionary is an alphabetically sorted list of all different words (i.e. strings of characters between given delimiters) of T, each word W being flanked by a varying amount of relevant information, such as the number of occurrences of W in T, the number of different documents in which W occurs, and, sometimes, linguistic details or morphological analyses of W, etc. At the very least, the record for W will have a pointer to the corresponding record in the references file, where references to all the occurrences of W in T are given. These references can be either *complete*, that is, listing for each occurrence, the document, paragraph, sentence and word numbers of that occurrence (in which case all the search process is done by manipulating lists of coordinates from the inverted files), or *partial*, listing only the document numbers of the occurrences of W (in which case part of the search process has to be done by processing the running text of some of these documents). Either way, once the correct set of documents is located, the text file is accessed to print or display relevant passages, as directed by the user.

3. THE PROBLEM: PROCESSING TRUNCATED TERMS

Several problems—linguistic, semantic, conceptual—are usually associated with the retrieval scheme described above. Most of them originate from the complex nature and unpredictable behaviour of free natural language textual data on the one hand, and from the necessity of reducing the query's topic, usually a concept or an idea, to a set of strings of characters combined by various operators on the other. Here we address ourselves to the very common problem of formal linguistic variants—more precisely: prefix, suffix and infix variants—of a given query term.

Suppose for example that a user wants to retrieve from a given database all documents that mention computers and computing. Ignoring for the time being the problem of synonymy, he has to locate (or ask the computer to locate) among other terms all occurrences of the different variants of the verb *compute* and the noun *computer*. These will include, for example, *compute*, *computes*, *computing*, *computed*, *computer*, *computers*, etc. and maybe also *computor's*, *computerize*, etc. Since however all of these forms begin with the same kernel COMPUT (and on the other hand any term beginning with COMPUT is most certainly pertinent), a handy way for the user to frame his query is to submit the truncated query term COMPUT*, where "*" is a *variable length don't care character* (VLDC) that stands for an arbitrary (possibly empty) string. In other cases the VLDC will appear as a prefix, as for example in *MAGNETISM, which will stand for DIAMAGNETISM, PARAMAGNETISM, etc. This feature and the more complex one in which VLDCs appear both as prefix and suffix, are most useful for technical literature (chemistry, botany, medicine, etc.), where for example *MYCIN* will capture the names of a large number of antibiotics, *OXYD* a large number of chemical processes, etc. For highly inflected languages such as Hebrew or Arabic, where prepositions and other items (such as: *the*, *to*, *from*, *in*, *and*,...) can be prefixed to nouns and verbs, and possessives (*mine*, *yours*,...) can be suffixed to them, the option of having VLDCs added as prefixes and suffixes (simultaneously) to query terms is an absolute necessity. Even in non-technical English such a feature can sometimes be useful, as is shown by the following example (adapted from [14]):

work	works	work's	works'	worked
working	worker	workers	worker's	workers'
rework	reworks	reworked	reworking	reworkable
workable	unworkable			

Finally a VLDC appearing as an infix character can be very useful for representing foreign names (BA*TYAR, the * standing for H, K, KH, CH, SH, SCH, etc.) or chemical compounds.

To sum up, we would like to give the user the option of including in his query truncated terms of the form X^*, $^*X$, $^*X^*$, and X^*Y, as well as the ordinary X. (Here and throughout the paper W, X, Y, Z stand for specified strings of characters; $|X|$ is the length of X). Before going on with the search, the system has first to find out—using appropriate inverted files—the set of words in the text that are represented by such a truncated term. This is straightforward for suffix truncated terms X^*, since all one has to do is to access directly the first record in the dictionary whose key *begins* with X, reading sequentially all subsequent records up to the first one that does not satisfy this criterion.

This approach, and the dictionary file, are of course of no direct use for the prefix truncated term $^*X$. Discussing this question in a 1977 paper [7], the authors say: "However, the processing of prefixes is not as simple. In this case only the ending of a word is known, while the file is organized by word beginnings. The conventional solution is to search the file sequentially for all terms which match the specified substring". Clearly, however, this is not a practical solution since dictionary files, even for moderate databases, can be of the order of a hundred thousand entries. A much more efficient solution would be to duplicate the dictionary, writing all the words in reverse order (i.e. writing $X = x_1 \ldots x_n$ in the form $x_n \ldots x_1$) and then sorting alphabetically, and processing the prefixed term $y_1 \ldots y_k$ by looking up its reversed form $y_k \ldots y_1$ in the reverse dictionary, thus bringing the problem back to the suffix case. (This approach was implemented in the RESPONSA system—for whose Hebrew database this option is an absolute must—as early as 1968). There is a certain cost to pay for overhead disk storage, but as will be argued later, it is negligible.

Things however become more complicated with truncated terms of the form $^*X^*$ or X^*Y. Quoting again from a 1978 paper on full-text systems [2], we reproduce the authors' conclusion after their description of "don't-care characters" and their pattern matching problems: "the question of whether file inversion techniques are feasible in systems which require the full generality of word pattern matching as outlined above remains open". (See also the interesting and typical note [13].) Finally at the 1980 Symposium on Research and Development in

Information Retrieval, two propers on full-text systems were delivered in which this problem of processing truncated terms was mentioned as a major obstacle when implementing such systems in practical situations.

Finding an adequate set of files and algorithms for processing such terms (and truncated terms in general) is then the subject of this note. For the sake of clarity, we first present the basic idea in its simplest form, adding refinements and variants only latter on.

4. THE BASIC IDEA

The main features of the solution to be presented are as follows:

(a) A *uniform* approach is given for processing all truncated query terms of the form:

$$X,\ X^*,\ {}^*X,\ {}^*X^*,\ X^*Y.$$

(b) All the required words are found and retrieved essentially by one disk access and with practically negligible computing time. The method is thus well-suited for on-line application.

(c) The algorithms process equally well truncated terms X of any length, including terms of length 1 or 2.

(d) The procedures are easily adaptable to other slight variants of VLDC, such as fixed-length (or bounded-length) don't-care characters, don't-care characters restricted by tables, etc.

(e) The required files and algorithms have a particularly transparent structure, and their implementation is easy and straightforward.

(f) The overhead storage required by this approach, although not totally negligible, can be kept within practical and reasonable bounds by using various text-compression techniques.

(g) Although we shall not pursue the point in this paper, the constructed files can be used for a variety of other applications, in particular as tools for text-compression procedures.

Suppose then we are given a text T and let D be the dictionary file of T. D contains a record for every different word of T. For our purposes we can assume that the record contains only the word itself, which is in fact the record key.

Using D, a new file PD, the (cyclically) *permuted dictionary* is constructed by the following procedure:

(a) Append to the end of each word $X = x_1 \ldots x_n$ in D a special termination character—say "/"—that does not belong to the original alphabet, to get the *augmented word* $X = x_1 \ldots x_n/$.

(b) For every augmented word $x_1 \ldots x_n/$ construct $n + 1$ variants by cyclically shifting the word—wrapping it around itself—$1, 2, \ldots n + 1$ characters. The original augmented $X/$ thus generates the variants:

$$/x_1 \ldots x_n,\ x_n/x_1 \ldots x_{n-1},\ x_{n-1}x_n/x_1 \ldots x_{n-2},\ x_1 \ldots x_n/.$$

(c) Pad each resulting word with at least one blank.

(d) Sort the resulting file alphabetically using the sort sequence: blank, $/$, $a, \ldots, z$.

This is the required file that supersedes D.

Figure 1 gives an example of the construction of PD for a dictionary of just three words: *ABC, BABC, BCAB.*

In order to specify the algorithms that process truncated terms using the permuted dictionary, we now assume that a procedure GETP is available which, given a key Z, directly accesses the first record whose key *begins* with Z (if such a record exists) and retrieves it and all subsequent records whose keys also begin with Z. GETP is readily implemented as a slight variant of standard file-access procedures. Processing the truncated terms consists now of simply calling GETP with the appropriate key as follows:

(a) Query term X : GETP $/Xb$ or X/b
(b) Query term X^* : GETP $/X$
(c) Query term ${}^*X$: GETP $X/$
(d) Query term ${}^*X^*$: GETP X
(e) Query term X^*Y : GETP Y/X.

Case (a) is obvious. For (b) we note that the retrieved words are already sorted alphabetically. In fact, as is easily checked (see also Fig. 1), the permuted dictionary contains at its very beginning a complete copy of the original alphabetically sorted one. This is a useful feature, since many systems give the user the option of submitting a word and receiving a "page" of the dictionary that begins with that word, browsing freely in the preceding and following dictionary pages. Also it allows for a printed listing of the alphabetical dictionary by just one pass over the initial section of PD.

For (c) if the retrieved words are $X/Y_1, \ldots, X/Y_n$, then the required words are $Y_1X, \ldots, Y_nX$. Referring to Fig. 1, if we need for example all terms represented by $*C$, we call GETP with the key $C/$ to retrieve C/AB and C/BAB, i.e. the text words ABC and $BABC$. Accidentally, the words retrieved in this case are sorted, but this is not always so. The retrieved words $Z_i = Y_iX$ will be sorted by their prefixes Y_i, but because of the different lengths of Y_i this does not always induce an alphabetical order on the Z_i (retrieving for example Z/A, Z/AB gives AZ, ABZ).

If the retrieved words are to be displayed for the user, or used to access the postings file, they should ideally be in alphabetical order for convenience and efficiency. However if they are partially sorted by their prefixes, this may be sufficient. The user will not be severely inconvenienced if the order is not absolutely correct, and on the other hand if the physical blocks on disk are of a reasonable size and properly buffered, the fact that keys are presented slightly out of order should not cost too many extra transfers. It is therefore probably not necessary to sort the words retrieved in case (c).

For case (d) if the retrieved strings are $XY_1/Z_1, \ldots, XY_n/Z_n$ then the required words are $Z_1XY_1, \ldots, Z_nXY_n$. This list of words is sorted by the suffixes Y_i as first key and the prefixes Z_i as the second one. There is little choice but to sort the retrieved words before presenting them to the user or looking them up in the postings file, if this is what is required. Note also that if a word W in the original dictionary contains k occurrences of X, then W will appear k times in the list. Thus asking for $*B*$ in the example of Fig. 1 will retrieve the strings:

$$B/BCA,\ BABC/,\ BC/A,\ BC/BA,\ BCAB/$$

which correspond to the list:

$$BCAB,\ BABC,\ ABC,\ BABC,\ BCAB$$

where each word occurs with its appropriate multiplicity. In practice however this will be a rather uncommon situation unless one is frequently processing truncated terms of length 1 or 2.

Finally for case (e) if the retrieved strings are $Y/XZ_1, \ldots, Y/XZ_n$, then the required words are $XZ_1Y, \ldots, XZ_nY$. These words are sorted by the prefixes XZ_i, but as in case (c), because of the different lengths of the Z_i, not necessarily sorted completely. The argument used above in case (c) applies with even more force, since the prefixes XZ_i will normally be reasonably long. Again, therefore, it is probably not worth sorting the retrieved words before presenting them to the user or accessing the postings file.

Incidentally, we note that the number of records read and checked by GETP is equal to the number of relevant strings in the text (plus one). Thus no time is spent on checking and discarding irrelevant items.

We have here therefore a uniform, easily implemented on-line procedure for processing query terms with or without VLDCs. The only apparent problem with this approach is the seemingly high overhead storage needed for the permuted dictionary. This issue will be discussed in Section 6. Here we merely mention that for very large databases, and under the worst conditions, the overhead storage needed for the permuted dictionary will be in the range of 2–3% of the total disk storage required by the retrieval system and, with some sophistication, it can even be lowered to less than 1%. This is certainly not a prohibitive cost to pay for such an option, if it is indeed needed.

5. REFINEMENTS AND VARIANTS

As already remarked, the permuted dictionary contains at its very beginning a copy of the original one, with all words preceded by the separator "/", and this part of the file can be used to process terms of the form X^*, to print a copy of the dictionary or to display selected pages from it on user request. To save storage, however, we can omit this part of the permuted dictionary (i.e. when producing the cyclically shifted variants of an augmented dictionary word $x_1 \ldots, x_n/$, we omit the variant $/x_1 \ldots x_n$). The processing of the terms $*X$, $*X*$, and $X*Y$ is not affected by this omission, since the corresponding keys for GETP—$X/$, X, and Y/X respectively—do not refer at all to that part of the file. The processing of terms of the form X and X^* can still be realized by noting that the set of all strings of the form $X/$ that are scattered throughout the permuted dictionary represent in fact another copy of the original one, again in the proper alphabetical order. Thus to process X^* or X one calls GETP X and retains from the retrieved list only those strings that *end* with "/" in the first case, or the first string retrieved by GETP (if it ends with "/") in the second. Similarly for printing a copy of the dictionary (or displaying a page that begins with X), we have only to sequentially read PD (or to access the record $X/$, respectively) outputting all strings that begin with X and end with "/".

Focussing especially on the function X^*, we notice that a certain processing time is now needed to find out for every string retrieved by GETP whether its last non-blank character is indeed "/". In order to save a substantial amount of this time overhead, we can preprocess PD by adding to every record a 1-bit flag that indicates whether the string terminates with a "/" or not. In certain cases, a somewhat more elaborate solution can be suggested, namely to add to every record a pair of numbers (L, M), where L is the length of the string including "/", and M gives the position of "/" in the string. This brings us to the *amended permuted dictionary* APD, where all strings of the form $/X$ are omitted from the file, the blank padding is dropped from the word, and the pair (L, M) is added to the record. Figure 1(e) gives the amended permuted dictionary for the dictionary of Fig. 1(a).

The processing of X and X^* with APD now requires that the strings retrieved by GETP with the keys $X/$ and X respectively be checked for the condition $L = M$. We note again that only for X^* has the checking to be done on all records retrieved by GETP (i.e. all records that begin with X). For the term X one stops of course after checking the first record retrieved.

The introduction of the pair (L, M) allows efficient processing of other types of don't-care characters, such as the *bounded-length don't-care character*, symbolised here by $(*n)$, meaning that the variable string has to have at most n characters. This operator can be processed according to the following scheme:

	Query term	*Key for GETP*	*Conditions to be checked*
(b)	$X(*n)$	X	$L = M,\ M - \lvert X\rvert \leq n + 1$
(c)	$(*n)X$	$X/$	$L - M \leq n$
(d)	$(*n)X(*m)$	X	$M - \lvert X\rvert \leq m + 1,\ L - M \leq n$
(e)	$X(*n)Y$	Y/X	$L - \lvert YX\rvert \leq n + 1.$

Another variant of the don't-care character is the *fixed-length don't-care*, which we symbolize by $(!n)$. The preceding scheme, with some obvious modifications, can also be used for processing the cases $X(!n)$, $(!n)X$, $(!n)X(!m)$, $X(!n)Y$. Still another special don't-care character—we denote it by #—is sometimes used to stand for any string from a predefined table of strings. Thus, in English, one would have a table of common endings and suffixes, such as *tion, ment, able, ing*, etc..., and (separately) of prefixes such as *un, re*, etc., so that *stand#* will not retrieve *standard*, nor *#play, splay*. (Such an option was implemented in the Responsa Project for the Hebrew language). Because of the (L, M) pair that gives the exact locations of prefix/suffix/infix, this feature too can be efficiently processed.

Of course, if for some particular application processing time is much more critical than disk storage, then one might add the (L, M) pair while leaving the initial section of the dictionary. In this case GETP need not read irrelevant records to be checked and discarded by the algorithm for terms of the form X^*. Since such terms are likely to be common, a considerable amount of time may be saved. The cost is likely to be about 15% extra storage overhead.

It can be argued at this point that since the pair (L, M) gives the exact location of "/", the occurrence of "/" in the string is actually redundant. Omitting it from the words in APD will not only save about 12% of the storage needed, but will sometimes turn two consecutive entries (of the form A/BC and AB/C, say) into equal strings, thus increasing the compression of the

				L M	L M n R
ABC	ABC/	/ABC	/ABC	-	
BABC	BABC/	C/AB	/BABC	-	
BCAB	BCAB/	BC/A	/BCAB	-	
		ABC/	AB/BC	5 3 AB/BC	5 3 0 AB/BC
		/BABC	ABC/	4 4 ABC/	4 4 2 C/
		C/BAB	ABC/B	5 4 ABC/B	5 4 4 B
		BC/BA	B/BCA	5 2 B/BCA	5 2 0 B/BCA
		ABC/B	BABC/	5 5 BABC/	5 5 1 ABC/
		BABC/	BC/A	4 3 BC/A	4 3 1 C/A
		/BCAB	BC/BA	5 3 BC/BA	5 3 3 BA
		B/BCA	BCAB/	5 5 BCAB/	5 5 2 AB/
		AB/BC	C/AB	4 2 C/AB	4 2 0 C/AB
		CAB/B	C/BAB	5 2 C/BAB	5 2 2 BAB
		BCAB/	CAB/B	5 4 CAB/B	5 4 1 AB/B
(a)	(b)	(c)	(d)	(e)	(f)

Fig. 1. The permuted dictionary. (a) The original dictionary D. (b) The augmented words. (c) Cyclically shifted augmented words. (d) The permuted dictionary PD (i.e. the sorted form of (c)). (e) The amended permuted dictionary APD (L is the length of each string; M is the position of the "/"). (f) The compressed APD (n is the number of characters copied from the preceding entry; R is the rest of the string).

permuted dictionary when using the methods to be described in the next section. It should be noticed, however, that the omission of "/" from the APD entries will completely destroy the alphabetical ordering of this file, and thus the possibility of directly accessing any of its entries using GETP. We were not able to devise a scheme that would enable us to omit "/" while assuring a simple, standard, and efficient direct access to the file.

The four patterns of truncated terms discussed above, where * appears as a prefix, suffix, infix, or both prefix and suffix, are the basic and most common forms to occur in ordinary queries. The permuted dictionary is capable, however, of processing more complex patterns, albeit somewhat less efficiently than the basic ones.

Take for example the case of $*X_1*X_2*\ldots*X_k*$. A possible approach is to construct the lists L_i of words that contain X_i (by processing $*X_i*$), to form their intersection L, and to check every word in L for the occurrence of $X_1, X_2, \ldots, X_k$ in this order. A better approach perhaps would be to "guess" which X_i has the least number of "parent words" in which it occurs, to produce the list K of these words, and for each X_iY/Z in this list such that $M-|X_i| \geq T_2$ and $L-M \geq T_1$, where $T_1=|X_1|+\cdots+|X_{i-1}|$ and $T_2=|X_{i+1}|+\cdots+|X_k|+1$, to check Y for the occurrence of $X_{i+1}, \ldots, Xk$ and Z for the occurrence of $X_1, \ldots, X_{i-1}$. A reasonable guess is to choose an X_i with greatest length; if all X_i have lengths 1 or 2, then one can use a prestored table of a few hundred entries in which the number of parent words for every such string (of length 1 or 2) is listed, choosing the element with lowest frequency as the pivot element X_i.

Although the need for such a complex pattern is quite infrequent for English databases, it was very pressing in the Responsa system, because of the complex morphological nature of Hebrew in general and of the Responsa Rabbinical Hebrew in particular. An altogether different, extremely powerful, and quite involved technique, designed and implemented by Eliezer Segal N"I under the direction of the second author, has been implemented and is operational on-line (with almost instantaneous response) since 1979. For a database of 30,000 documents, 30 million words, and half-a-million dictionary entries, the overhead storage was less than 8 million bytes. The details will appear elsewhere.

6. COMPRESSION METHODS

We now discuss the problem of the overhead storage required to implement the permuted dictionary. First we note that if the mean length of word-types in the database is k, then, taking into account the separator "/", the number of entries in the permuted dictionary will be $(k+1)$ times that of the original. k is usually typical of the natural language under consideration; it is about 8 for English [10]. Of course, the mean length of entries in the permuted dictionary will not be exactly equal to that of the words in the original. Usually it will be somewhat larger, since a word of length k in the original will generate $k+1$ entries of length $k+1$ in the permuted dictionary, i.e. long words generate proportionately more entries. Using the figures given in [10], we may calculate that in English mean word length in the permuted dictionary will be almost exactly 10 characters.

Moreover we quote from [2] the following numbers as characteristic of very large document retrieval systems:

Number of documents	1 million
Number of word-tokens	2 billion
Number of word-types	.5 million
Number of occurrences of the 100 most frequent words	1 billion.

Assuming an average of 5 characters (including delimiters) for word-token length ([10]), we get an uncompressed text-file of 10 billion characters. Assuming a minimal postings file with partial references (pointers to document numbers only), and with omission of the references to high frequency words, we still need 1 billion records of 3 bytes each at least, which brings the total uncompressed storage to 13 billion bytes. Assuming further a high compression ratio of 70% for these files, we are still left with about 4 billion bytes of disk storage. On the other hand, suppose we assume that, on average, each record of the dictionary file consists of 8 characters for the word and 16 for the related information (this is in fact quite a high figure). The permuted dictionary will have about 9 times as many records, and the average number of characters for a word will be about 10; add 2 bytes for L and M, and keep the other information unchanged. Thus we get 12 million bytes for the original dictionary, and 126 million bytes for the permuted one. The overhead then is 114 million bytes, or well under 3% of the total required storage. In reality, however, the postings file will often contain full references, sometimes even to the high frequency words (as in the Responsa project), the compression ratio is not so high, and the permuted file can certainly also be compressed, all factors bringing, plausibly, the overhead percentage to well under 1% of the total storage; which is a reasonable overhead for such an option.

The amended permuted dictionary is of course an ordered list of words, and thus can be compressed using any one of the multitude of text compression techniques developed in the past twenty years. (For a comprehensive bibliography on such techniques see [3]; [12] gives references to the relevant literature in the period 1975–1979).

A compression ratio of about 40% is quite common, and ratios of as much as 70% have been claimed in certain cases.

The method that seems to us best suited to the APD file, is an adaptation of an idea of A. Fraenkel suggested by him several years ago for the compression of the postings file of the Responsa system. Although simple, natural, and straightforward, the method gives a very good compression ratio without requiring any code tables or elaborate decompression techniques; curiously, however, it is seldom mentioned or used in the literature. It is based on the observation that in a dictionary of a natural language text, two consecutive entries will usually have a few leading letters in common; we can therefore eliminate these letters from the second entry, while adding to it the number of letters eliminated and to be copied from the previous entry. More formally, let X_i, $1 \leq i \leq N$ be the entries in the original file, and Y_i the corresponding entries in the compressed one. Then each Y_i consists of a pair (n_i, R_i), where n_i is a number (the *copy* factor) and R_i a string (the *residue*), defined as follows:

(1) $n_1 = 0 \;\; R_1 = X_1$

(2) Let m be the number of equal leading letters in X_i and X_{i+1}, $1 \leq i < N$. Then $n_{i+1} = m$, and R_{i+1} is X_{i+1} with the first m letters removed. (Note that R_i can never be empty, since APD is an alphabetical dictionary which never contains two consecutive, exactly equal entries.)

Figure 1(f) gives the compressed form of the list 1(e) obtained by this method.

The compression algorithm is immediate; the only point that should be emphasized is that the copy factor always refers to the *uncompressed* form of the previous entry. If the compressed file is to be processed sequentially then the decompression algorithm is also easy.

Let P and Q be string variables initialized to blanks; the following steps are performed sequentially on all the entries Y_i of the compressed file, to reconstruct X_i:

(1) Let Q = first n_i letters of P.
(2) Let Q = concatenation of Q with R_i.
(3) $P \leftarrow Q$; $X_i \leftarrow Q$.

Usually, however, direct access is also needed to any entry of the dictionary, as is the case with the APD. To this end, the compressed file records are grouped into buckets of fixed size, which can conveniently be chosen equal to the physical size of the read/write block of the system; usually such a bucket will contain from a few hundred to a few thousand entries of the compressed file. Each bucket begins with the full non-compressed form of the corresponding entry; since the number of buckets will generally be a few hundred (certainly not more than 2 to 3 thousand), only a few thousand additional characters are required and this is an insignificant overhead. No need exists therefore from this point of view to optimize bucket size or the "cut point" between buckets. A table containing a list of the first entries of all buckets is constructed and kept in internal memory. Given any key Z, a binary search in this table—with at most 10 to 15 comparisons—will locate two consecutive entries R and S such that $R < Z \leq S$, thus giving the serial number of the bucket containing the first entry that begins with Z, and this bucket B is now read into memory.

One has now, it seems, to decompress B and to compare Z sequentially with the decompressed entries of B to locate the first relevant one. This is not so, however; indeed this entry can be located simply by processing the copy factors n_i, with only occasional decompression. Moreover, once the first relevant entry is located, there is no need to check all subsequent entries, character by character, to retrieve those that begin with Z; a check of the copy factor is sufficient. There follows the algorithm for GETP which, given a key Z and a bucket B in the memory, will process it to output all entries in B beginning with Z. (The entries in B will be denoted by (n_i, R_i); note that $n_1 = 0$.)

(1) Set $i = 1$. Set $X_1 = R_1$.

(2) Let m be the number of equal leading letters in Z and X_i.

(3) If $m = |Z|$, output X_i; decompress and output all subsequent entries for which the copy factor $n \geq |Z|$, stopping at the first entry for which $n < |Z|$.

(4) Otherwise, repeatedly increment i by 1 until $n_i \leq m$. Now X_i is the first n_i letters of Z concatenated with R_i.

(5) Go to 2.

For example, suppose that we read the compressed dictionary of Fig. 1(f) with key C; suppose also that all the records are in the same bucket. Then we read item 1, *AB*/*BC*, which is already decompressed; skip items 2 and 3, which have $n_i > 0$; decompress item 4, *B*/*BCA*; skip the next four items, again because $n_i > 0$; and finally decompress *C*/*AB* and carry on reading from there.

The compression technique adopted will therefore save not only storage space but also processing time.

One can push the compression a little further with very small processing time overhead by the following observation. It is reasonable to assume that many of the residues in the compressed APD will be common endings, such as *-ing*, *-able*, *-ed*, *-ers*, *-tion*, *-ment*, or, since the dictionary is permuted, common prefixes such as *un-*, *re-*, *over-*, and will appear quite frequently in the file (scattered in different places). One can save space therefore by substituting short fixed-length codes for some of these residues. The residues to be coded should of course be chosen to maximize the storage compression gain; on the other hand, a scheme has to be devised to recognise and process non-coded residues. We shall not go into this topic in more detail here; the interested reader should consult the literature listed in the references.

REFERENCES

[1] BASIS, Battelle, Columbus laboratories, Columbus, Ohio.
[2] R. M. Bird, J. B. Newsbaum and J. L. Trefftzs, Text file inversion: an evaluation, *Proc. Fourth Workshop for Computer Architecture for Non-Numeric Processing*, pp. 42–50. Syracuse University, (Aug. 1978).

[3] B. CARRIGAN, Data compression (Citations from NTIS data base), PB80–801632, NTIS, Springfield, Virginia (Dec. 1979).

[4] DOCU/MASTER, The key to information storage and retrieval, Turnkey Systems, Norwalk, Connecticut.

[5] M. ANNE FOSTER and SHIRLEY A. LOUNDER, *Applications of computer technology to law* (1969–1978): *A selected bibliography*, Working Paper No. 4. Canadian Law Information Council, Ottawa, Canada (1980).

[6] RICHARD H. GIERING, DATA/CENTRAL, Technical Specifications DGA-75-2, 1975, D. G. Associates, Dayton, Ohio.

[7] L. A. HOLLAAR and W. H. STELLHORN, A specialized architecture for textual information retrieval, *Proc. AFIPS* 1977, *NCC*, Vol. 46, AFIPS Press, Montvale, New Jersey pp. 697–702.

[8] J. F. HORTY, *Searching statutory laws by computer*, Interim Rep. No. 2. Health Law Center, Univ. of Pittsburgh, Pittsburgh, Pennsylvania, May (1962).

[9] C. KEREN, H. E. SEELBACH and P. WOLLMAN, Using Minicomputers in information work-project DOMESTIC, *Nachr. f. Dokum.* 1978, **29**, 163.

[10] HENRY KUCERA and NELSON W. FRANCIS, *Computational Analysis of Present-day American English*. Brown Universty Press, Providence, Rhode Island (1967).

[11] JOHN KEITH MORGAN, *Description of an experimental, on-line, minicomputer-based information retrieval system*. Rep. No UIUCDCS-R-76-779. Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, Illinois (Feb. 1976).

[12] T. RADHAKRISHNAN and P. LAPLANTE, *A selected bibliography on compression of signal, text, and image data*. Dept. of Computer Science, Concordia University, Montreal, Canada (May 1980).

[13] G. SALTON, Chairman's message. FORUM, *SIGIR Newsletter* 1980, XIV(3), 6.

[14] EUGÈNE S. SCHWARTZ, A dictionary for minimum redundancy encoding, *J. Ass. Comp. Mach.* 1963, **10**, 413.

[15] Storage and information retrieval system/Virtual storage (STAIRS/VS). Program Product 5740–XR1, IBM World Trade Corporation (1974).