Chapter 7
Text Operations

with Nivio Ziviani

7.1 Introduction

As discussed in Chapter 2, not all words are equally significant for representing the semantics of a document. In written language, some words carry more *meaning* than others. Usually, *noun* words (or groups of noun words) are the ones which are most representative of a document content. Therefore, it is usually considered worthwhile to preprocess the text of the documents in the collection to determine the terms to be used as *index terms*. During this preprocessing phase other useful text operations can be performed such as elimination of stopwords, stemming (reduction of a word to its grammatical root), the building of a thesaurus, and compression. Such text operations are discussed in this chapter.

We already know that representing documents by sets of index terms leads to a rather imprecise representation of the semantics of the documents in the collection. For instance, a term like '*the*' has no meaning whatsoever by itself and might lead to the retrieval of various documents which are unrelated to the present user query. We say that using the set of all words in a collection to index its documents generates too much *noise* for the retrieval task. One way to reduce this noise is to reduce the set of words which can be used to refer to (i.e., to index) documents. Thus, the preprocessing of the documents in the collection might be viewed simply as a process of controlling the size of the vocabulary (i.e., the number of distinct words used as an index terms). It is expected that the use of a controlled vocabulary leads to an improvement in retrieval performance.

While controlling the size of the vocabulary is a common technique with commercial systems, it does introduce an additional step in the indexing process which is frequently not easily perceived by the users. As a result, a common user might be surprised with some of the documents retrieved and with the absence of other documents which he expected to see. For instance, he might remember that a certain document contains the string '*the house of the lord*' and notice that such a document is not present among the top 20 documents retrieved in

response to his query request (because the controlled vocabulary contains neither 'the' nor 'of'). Thus, it should be clear that, despite a potential improvement in retrieval performance, text transformations done at preprocessing time might make it more difficult for the user to interpret the retrieval task. In recognition of this problem, some search engines in the Web are giving up text operations entirely and simply indexing all the words in the text. The idea is that, despite a more noisy index, the retrieval task is simpler (it can be interpreted as a full text search) and more intuitive to a common user.

Besides document preprocessing, other types of operations on documents can also be attempted with the aim of improving retrieval performance. Among these we distinguish the construction of a thesaurus representing conceptual term relationships and the clustering of related documents. Thesauri are also covered in this chapter. The discussion on document clustering is covered in Chapter 5 because it is an operation which might depend on the current user query.

Text normalization and the building of a thesaurus are strategies aimed at improving the precision of the documents retrieved. However, in the current world of very large digital libraries, improving the efficiency (in terms of time) of the retrieval process has also become quite critical. In fact, Web search engines are currently more concerned with reducing query response time than with improving precision and recall figures. The reason is that they depend on processing a high number of queries per unit of time for economic survival. To reduce query response time, one might consider the utilization of text compression as a promising alternative.

A good compression algorithm is able to reduce the text to 30–35% of its original size. Thus, compressed text requires less storage space and takes less time to be transmitted over a communication link. The main disadvantage is the time spent compressing and decompressing the text. Until recently, it was generally understood that compression does not provide substantial gains in processing time because the extra time spent compressing/decompressing text would offset any gains in operating with compressed data. Further, the use of compression makes the overall design and implementation of the information system more complex. However, modern compression techniques are slowly changing this understanding towards a more favorable view of the adoption of compression techniques. By modern compression techniques we mean good compression and decompression speeds, fast random access without the need to decode the compressed text from the beginning, and direct searching on the compressed text without decompressing it, among others.

Besides compression, another operation on text which is becoming more and more important is *encryption*. In fact, due to the fast popularization of services in the Web (including all types of electronic commerce), key (and old) questions regarding security and privacy have surfaced again. More than ever before, impersonation and unauthorized access might result in great prejudice and financial damage to people and organizations. The solution to these problems is not simple but can benefit from the operation of encrypting text. Discussing encrypted text is beyond the scope of this book but an objective and brief introduction to the topic can be found in [501].

In this chapter, we first discuss five preprocessing text operations including thesauri. Following that, we very briefly summarize the problem of document clustering (which is discussed in detail in Chapter 5). Finally, a thorough discussion on the issue of text compression, its modern variations, and its main implications is provided.

7.2 Document Preprocessing

Document preprocessing is a procedure which can be divided mainly into five text operations (or transformations):

(1) Lexical analysis of the text with the objective of treating digits, hyphens, punctuation marks, and the case of letters.

(2) Elimination of stopwords with the objective of filtering out words with very low discrimination values for retrieval purposes.

(3) Stemming of the remaining words with the objective of removing affixes (i.e., prefixes and suffixes) and allowing the retrieval of documents containing syntactic variations of query terms (e.g., connect, connecting, connected, etc).

(4) Selection of index terms to determine which words/stems (or groups of words) will be used as an indexing elements. Usually, the decision on whether a particular word will be used as an index term is related to the syntactic nature of the word. In fact, noun words frequently carry more semantics than adjectives, adverbs, and verbs.

(5) Construction of term categorization structures such as a thesaurus, or extraction of structure directly represented in the text, for allowing the expansion of the original query with related terms (a usually useful procedure).

In the following, each of these phases is discussed in detail. But, before proceeding, let us take a look at the logical view of the documents which results after each of the above phases is completed. Figure 1.2 is repeated here for convenience as Figure 7.1. As already discussed, by aggregating the preprocessing phases, we are able to move the logical view of the documents (adopted by the system) from that of a full text to that of a set of high level indexing terms.

7.2.1 Lexical Analysis of the Text

Lexical analysis is the process of converting a stream of characters (the text of the documents) into a stream of words (the candidate words to be adopted as index terms). Thus, one of the major objectives of the lexical analysis phase is the identification of the words in the text. At first glance, all that seems to be involved is the recognition of spaces as word separators (in which case, multiple

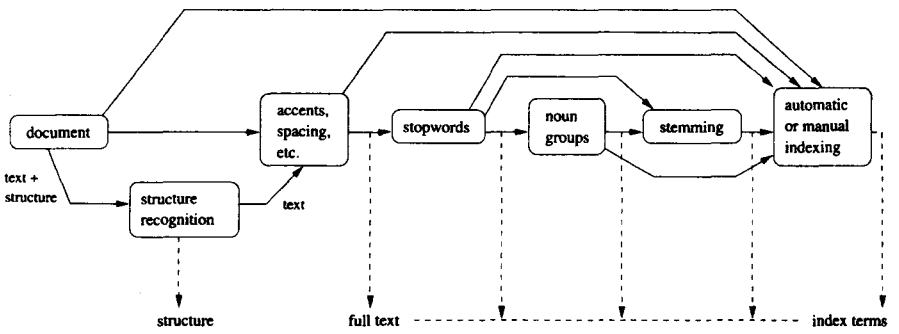


Figure 7.1 Logical view of a document throughout the various phases of text preprocessing.

spaces are reduced to one space). However, there is more to it than this. For instance, the following four particular cases have to be considered with care [263]: digits, hyphens, punctuation marks, and the case of the letters (lower and upper case).

Numbers are usually not good index terms because, without a surrounding context, they are inherently vague. For instance, consider that a user is interested in documents about the number of deaths due to car accidents between the years 1910 and 1989. Such a request could be specified as the set of index terms {deaths, car, accidents, years, 1910, 1989}. However, the presence of the numbers 1910 and 1989 in the query could lead to the retrieval, for instance, of a variety of documents which refer to either of these two years. The problem is that numbers by themselves are just too vague. Thus, in general it is wise to disregard numbers as index terms. However, we have also to consider that digits might appear mixed within a word. For instance, '510B.C.' is a clearly important index term. In this case, it is not clear what rule should be applied. Furthermore, a sequence of 16 digits identifying a credit card number might be highly relevant in a given context and, in this case, should be considered as an index term. A preliminary approach for treating digits in the text might be to remove all words containing sequences of digits unless specified otherwise (through regular expressions). Further, an advanced lexical analysis procedure might perform some date and number normalization to unify formats.

Hyphens pose another difficult decision to the lexical analyzer. Breaking up hyphenated words might be useful due to inconsistency of usage. For instance, this allows treating 'state-of-the-art' and 'state of the art' identically. However, there are words which include hyphens as an integral part. For instance, gilt-edge, B-49, etc. Again, the most suitable procedure seems to adopt a general rule and specify the exceptions on a case by case basis.

Normally, punctuation marks are removed entirely in the process of lexical analysis. While some punctuation marks are an integral part of the word (for

instance, '510B.C.'), removing them does not seem to have an impact in retrieval performance because the risk of misinterpretation in this case is minimal. In fact, if the user specifies '510B.C' in his query, removal of the dot both in the query term and in the documents will not affect retrieval. However, very particular scenarios might again require the preparation of a list of exceptions. For instance, if a portion of a program code appears in the text, it might be wise to distinguish between the variables 'x.id' and 'xid.' In this case, the dot mark should not be removed.

The case of letters is usually not important for the identification of index terms. As a result, the lexical analyzer normally converts all the text to either lower or upper case. However, once more, very particular scenarios might require the distinction to be made. For instance, when looking for documents which describe details about the command language of a Unix-like operating system, the user might explicitly desire the non-conversion of upper cases because this is the convention in the operating system. Further, part of the semantics might be lost due to case conversion. For instance, the words *Bank* and *bank* have different meanings — a fact common to many other pairs of words.

As pointed out by Fox [263], all these text operations can be implemented without difficulty. However, careful thought should be given to each one of them because they might have a profound impact at document retrieval time. This is particularly worrisome in those situations in which the user finds it difficult to understand what the indexing strategy is doing. Unfortunately, there is no clear solution to this problem. As already mentioned, some Web search engines are opting for avoiding text operations altogether because this simplifies the interpretation the user has of the retrieval task. Whether this strategy will be the one of choice in the long term remains to be seen.

7.2.2 Elimination of Stopwords

As discussed in Chapter 2, words which are too frequent among the documents in the collection are not good discriminators. In fact, a word which occurs in 80% of the documents in the collection is useless for purposes of retrieval. Such words are frequently referred to as *stopwords* and are normally filtered out as potential index terms. Articles, prepositions, and conjunctions are natural candidates for a list of stopwords.

Elimination of stopwords has an additional important benefit. It reduces the size of the indexing structure considerably. In fact, it is typical to obtain a compression in the size of the indexing structure (for instance, in the size of an inverted list, see Chapter 8) of 40% or more solely with the elimination of stopwords.

Since stopword elimination also provides for compression of the indexing structure, the list of stopwords might be extended to include words other than articles, prepositions, and conjunctions. For instance, some verbs, adverbs, and adjectives could be treated as stopwords. In [275], a list of 425 stopwords is illustrated. Programs in C for lexical analysis are also provided.

Despite these benefits, elimination of stopwords might reduce recall. For instance, consider a user who is looking for documents containing the phrase '*to be or not to be.*' Elimination of stopwords might leave only the term *be* making it almost impossible to properly recognize the documents which contain the phrase specified. This is one additional reason for the adoption of a full text index (i.e., insert all words in the collection into the inverted file) by some Web search engines.

7.2.3 Stemming

Frequently, the user specifies a word in a query but only a variant of this word is present in a relevant document. Plurals, gerund forms, and past tense suffixes are examples of syntactical variations which prevent a perfect match between a query word and a respective document word. This problem can be partially overcome with the substitution of the words by their respective stems.

A *stem* is the portion of a word which is left after the removal of its affixes (i.e., prefixes and suffixes). A typical example of a stem is the word *connect* which is the stem for the variants *connected, connecting, connection,* and *connections.* Stems are thought to be useful for improving retrieval performance because they reduce variants of the same root word to a common concept. Furthermore, stemming has the secondary effect of reducing the size of the indexing structure because the number of distinct index terms is reduced.

While the argument supporting stemming seems sensible, there is controversy in the literature about the benefits of stemming for retrieval performance. In fact, different studies lead to rather conflicting conclusions. Frakes [275] compares eight distinct studies on the potential benefits of stemming. While he favors the usage of stemming, the results of the eight experimental studies he investigated do not allow us to reach a satisfactory conclusion. As a result of these doubts, many Web search engines do not adopt any stemming algorithm whatsoever.

Frakes distinguishes four types of stemming strategies: affix removal, table lookup, successor variety, and n-grams. Table lookup consists simply of looking for the stem of a word in a table. It is a simple procedure but one which is dependent on data on stems for the whole language. Since such data is not readily available and might require considerable storage space, this type of stemming algorithm might not be practical. Successor variety stemming is based on the determination of morpheme boundaries, uses knowledge from structural linguistics, and is more complex than affix removal stemming algorithms. N-grams stemming is based on the identification of digrams and trigrams and is more a term clustering procedure than a stemming one. Affix removal stemming is intuitive, simple, and can be implemented efficiently. Thus, in the remainder of this section we concentrate our discussion on algorithms for affix removal stemming only.

In affix removal, the most important part is suffix removal because most variants of a word are generated by the introduction of suffixes (instead of pre-

fixes). While there are three or four well known suffix removal algorithms, the most popular one is that by Porter because of its simplicity and elegance. Despite being simpler, the Porter algorithm yields results comparable to those of the more sophisticated algorithms.

The Porter algorithm uses a suffix list for suffix stripping. The idea is to apply a series of rules to the suffixes of the words in the text. For instance, the rule

$$s \longrightarrow \phi \tag{7.1}$$

is used to convert plural forms into their respective singular forms by substituting the letter s by nil. Notice that to identify the suffix we must examine the last letters in the word. Furthermore, we look for the longest sequence of letters which matches the left hand side in a set of rules. Thus, application of the two following rules

$$\begin{aligned} sses &\longrightarrow ss \\ s &\longrightarrow \phi \end{aligned} \tag{7.2}$$

to the word *stresses* yields the stem *stress* instead of the stem *stresse*. By separating such rules into five distinct phases, the Porter algorithm is able to provide effective stemming while running fast. A detailed description of the Porter algorithm can be found in the appendix.

7.2.4 Index Terms Selection

If a full text representation of the text is adopted then all words in the text are used as index terms. The alternative is to adopt a more abstract view in which not all words are used as index terms. This implies that the set of terms used as indices must be selected. In the area of bibliographic sciences, such a selection of index terms is usually done by a specialist. An alternative approach is to select candidates for index terms automatically.

Distinct automatic approaches for selecting index terms can be used. A good approach is the identification of noun groups (as done in the Inquery system [122]) which we now discuss.

A sentence in natural language text is usually composed of nouns, pronouns, articles, verbs, adjectives, adverbs, and connectives. While the words in each grammatical class are used with a particular purpose, it can be argued that most of the semantics is carried by the noun words. Thus, an intuitively promising strategy for selecting index terms automatically is to use the nouns in the text. This can be done through the systematic elimination of verbs, adjectives, adverbs, connectives, articles, and pronouns.

Since it is common to combine two or three nouns in a single component (e.g., *computer science*), it makes sense to cluster nouns which appear nearby in the text into a single indexing component (or concept). Thus, instead of simply

using nouns as index terms, we adopt noun groups. A *noun group* is a set of nouns whose syntactic distance in the text (measured in terms of number of words between two nouns) does not exceed a predefined threshold (for instance, 3).

When noun groups are adopted as indexing terms, we obtain a conceptual logical view of the documents in terms of sets of non-elementary index terms.

7.2.5 Thesauri

The word *thesaurus* has Greek and Latin origins and is used as a reference to a treasury of words [261]. In its simplest form, this treasury consists of (1) a precompiled list of important words in a given domain of knowledge and (2) for each word in this list, a set of related words. Related words are, in its most common variation, derived from a synonymity relationship.

In general, however, a thesaurus also involves some normalization of the vocabulary and includes a structure much more complex than a simple list of words and their synonyms. For instance, the popular thesaurus published by Peter Roget [679] also includes *phrases* which means that concepts more complex than single words are taken into account. Roget's thesaurus is of a general nature (i.e., not specific to a certain domain of knowledge) and organizes words and phrases in categories and subcategories.

An example of an entry in Roget's thesaurus is as follows:

> **cowardly** *adjective*
> Ignobly lacking in courage: *cowardly turncoats.*
> **Syns:** chicken (slang), chicken-hearted, craven, dastardly, faint-hearted, gutless, lily-livered, pusillanimous, unmanly, yellow (slang), yellow-bellied (slang).

To the adjective *cowardly*, Roget's thesaurus associates several synonyms which compose a thesaurus class. While Roget's thesaurus is of a generic nature, a thesaurus can be specific to a certain domain of knowledge. For instance, the Thesaurus of Engineering and Scientific Terms covers concepts related to engineering and technical terminology.

According to Foskett [261], the main purposes of a thesaurus are basically: (a) to provide a standard vocabulary (or system of references) for indexing and searching; (b) to assist users with locating terms for proper query formulation; and (c) to provide classified hierarchies that allow the broadening and narrowing of the current query request according to the needs of the user. In this section, however, we do not discuss how to use a thesaurus for modifying the user query. This issue is covered on Chapter 5 which also discusses algorithms for automatic construction of thesauri.

Notice that the motivation for building a thesaurus is based on the fundamental idea of using a *controlled vocabulary* for the indexing and searching. A controlled vocabulary presents important advantages such as normalization

of indexing concepts, reduction of noise, identification of indexing terms with a clear semantic meaning, and retrieval based on concepts rather than on words. Such advantages are particularly important in specific domains, such as the medical domain for which there is already a large amount of knowledge compiled. For general domains, however, a well known body of knowledge which can be associated with the documents in the collection might not exist. The reasons might be that the document base is new, that it is too large, or that it changes very dynamically. This is exactly the case with the Web. Thus, it is not clear how useful a thesaurus is in the context of the Web. Despite that, the success of the search engine named 'Yahoo!' (see Chapter 13), which presents the user with a term classification hierarchy that can be used to reduce the space to be searched, suggests that thesaurus-based techniques might be quite useful even in the dynamic world of the Web.

It is still too early to reach a consensus on the advantages of a thesaurus for the Web. As a result, many search engines simply use *all* the words in all the documents as index terms (i.e., there is no notion of using the concepts of a controlled vocabulary for indexing and searching purposes). Whether thesaurus-based techniques will flourish in the context of the Web remains to be seen.

The main components of a thesaurus are its index terms, the relationships among the terms, and a layout design for these term relationships. Index terms and term relationships are covered below. The layout design for term relationships can be in the form of a list or in the form of a bi-dimensional display. Here, we consider only the more conventional layout structure based on a list and thus, do not further discuss the issue of layout of the terms in a thesaurus. A brief coverage of topics related to this problem can be found in Chapter 10. A more detailed discussion can be found in [261].

Theasurus Index Terms

The terms are the *indexing* components of the thesaurus. Usually, a term in a thesaurus is used to denote a *concept* which is the basic semantic unit for conveying ideas. Terms can be individual words, groups of words, or phrases, but most of them are single words. Further, terms are basically nouns because nouns are the most concrete part of speech. Terms can also be verbs in gerund form whenever they are used as nouns (for instance, *acting*, *teaching*, etc.).

Whenever a concept cannot be expressed by a single word, a group of words is used instead. For instance, many concepts are better expressed by a combination of an adjective with a noun. A typical example is *ballistic missiles*. In this case, indexing the compound term directly will yield an entry under *balistic* and no entry under *missiles* which is clearly inadequate. To avoid this problem, the compound term is usually modified to have the noun as the first word. For instance, we can change the compound term to *missiles, ballistic*.

We notice the use of the plural form *missiles* instead of the singular form *missile*. The reasoning is that a thesaurus represents classes of things and thus it is natural to prefer the plural form. However, the singular form is used for

compound terms which appear normally in the singular such as *body temperature*. Deciding between singular and plural is not always a simple matter.

Besides the term itself, frequently it is necessary to complement a thesaurus entry with a *definition* or an *explanation*. The reason is the need to specify the precise meanings of a term in the context of a particular thesaurus. For instance, the term *seal* has a meaning in the context of *marine animals* and a rather distinct meaning in the context of *documents*. In these cases, the definition might be preceded by a context explanation such as *seal (marine animals)* and *seal (documents)* [735].

Thesaurus Term Relationships

The set of terms related to a given thesaurus term is mostly composed of synonyms and near-synonyms. In addition to these, relationships can be induced by patterns of co-occurrence within documents. Such relationships are usually of a hierarchical nature and most often indicate broader (represented by BT) or narrower (represented by NT) related terms. However, the relationship might also be of a lateral or non-hierarchical nature. In this case, we simply say that the terms are related (represented by RT).

As discussed in Chapter 5, BT and NT relationships define a classification hierarchy where the broader term is associated with a class and its related narrower terms are associated with the instances of this class. Further, it might be that a narrower term is associated with two or more broader terms (which is not the most common case though). While BT and NT relationships can be identified in a fully automatic manner (i.e., without assistance from a human subject), dealing with RT relationships is much harder. One reason seems to be that RT relationships are dependent on the specific context and particular needs of the group of users and thus are difficult to identify without knowledge provided by specialists.

On the Use of Thesauri in IR

As described by Peter Roget [679, 261], a thesaurus is a classification scheme composed of words and phrases whose organization aims at facilitating the expression of ideas in written text. Thus, whenever a writer has a difficulty in finding the proper term to express an idea (a common occurrence in serious writing), he can use the thesaurus to obtain a better grasp on the fundamental semantics of terms related to his idea.

In the area of information retrieval, researchers have for many years conjectured and studied the usefulness of a thesaurus for helping with the query formation process. Whenever a user wants to retrieve a set of documents, he first builds up a conceptualization of what he is looking for. Such conceptualization is what we call his *information need*. Given the information need, the user still has to translate it into a query in the language of the IR system. This usually

means that a set of index terms has to be selected. However, since the collection might be vast and the user inexperienced, the selection of such *initial* terms might be erroneous and improper (a very common situation with the largely unknown and highly dynamic collection of documents and pages which compose the Web). In this case, reformulating the original query seems to be a promising course of action. Such a reformulation process usually implies expanding the original query with related terms. Thus, it seems natural to use a thesaurus for assisting the user with the search for related terms.

Unfortunately, this approach does not work well in general because the relationships captured in a thesaurus frequently are not valid in the local context of a given user query. One alternative is to determine thesaurus-like relationships at query time. Unfortunately, such an alternative is not attractive for Web search engines which cannot afford to spend a lot of time with the processing of individual queries. This and many other interesting issues related to the use of thesaurus-based techniques in IR are covered in Chapter 5.

7.3 Document Clustering

Document clustering is the operation of grouping together similar (or related) documents in classes. In this regard, document clustering is not really an operation on the text but an operation on the collection of documents.

The operation of clustering documents is usually of two types: global and local. In a global clustering strategy, the documents are grouped accordingly to their occurrence in the whole collection. In a local clustering strategy, the grouping of documents is affected by the context defined by the current query and its *local* set of retrieved documents.

Clustering methods are usually used in IR to transform the original query in an attempt to better represent the user information need. From this perspective, clustering is an operation which is more related to the transformation of the user query than to the transformation of the text of the documents. In this book, document clustering techniques are treated as query operations and thus, are covered in Chapter 5 (instead of here).

7.4 Text Compression

7.4.1 Motivation

Text compression is about finding ways to represent the text in fewer bits or bytes. The amount of space required to store text on computers can be reduced significantly using compression techniques. Compression methods create a reduced representation by identifying and using structures that exist in the text. From the compressed version, the original text can be reconstructed exactly.

Text compression is becoming an important issue in an information retrieval environment. The widespread use of digital libraries, office automation

systems, document databases, and the Web has led to an explosion of textual information available online. In this scenario, text compression appears as an attractive option for reducing costs associated with space requirements, input/output (I/O) overhead, and communication delays. The gain obtained from compressing text is that it requires less storage space, it takes less time to be transmitted over a communication link, and it takes less time to search directly the compressed text. The price paid is the time necessary to code and decode the text.

A major obstacle for storing text in compressed form is the need for IR systems to access text randomly. To access a given word in a compressed text, it is usually necessary to decode the entire text from the beginning until the desired word is reached. It could be argued that a large text could be divided into blocks that are compressed independently, thus allowing fast random access to each block. However, efficient compression methods need to process some text before making compression effective (usually more than 10 kilobytes). The smaller the blocks, the less effective compression is expected to be.

Our discussion here focuses on text compression methods which are suitable for use in an IR environment. For instance, a successful idea aimed at merging the requirements of compression algorithms and the needs of IR systems is to consider that the symbols to be compressed are words and not characters (character-based compression is the more conventional approach). Words are the atoms on which most IR systems are built. Moreover, it is now known that much better compression is achieved by taking words as symbols (instead of characters). Further, new word-based compression methods allow random access to words within the compressed text which is a critical issue for an IR system.

Besides the economy of space obtained by a compression method, there are other important characteristics to be considered such as compression and decompression speed. In some situations, decompression speed is more important than compression speed. For instance, this is the case with textual databases in which it is common to compress the text once and to read it many times from disk.

Another important characteristic of a compression method is the possibility of performing compressed pattern matching, defined as the task of performing pattern matching in a compressed text without decompressing it. In this case, sequential searching can be speeded up by compressing the search key rather than decoding the compressed text being searched. As a consequence, it is possible to search faster on compressed text because much less text has to be scanned. Chapter 8 presents efficient methods to deal with searching the compressed text directly.

When the text collection is large, efficient text retrieval requires specialized index techniques. A simple and popular indexing structure for text collections are the inverted files. Inverted files (see Chapter 8 for details) are especially adequate when the pattern to be searched for is formed by simple words. Since this is a common type of query (for instance, when searching the Web), inverted files are widely used for indexing large text collections.

An inverted file is typically composed of (a) a vector containing all the distinct words in the text collection (which is called the *vocabulary*) and (b) for

each word in the vocabulary, a list of all documents (identified by document numbers) in which that word occurs. Because each list of document numbers (within the inverted file) is organized in ascending order, specific compression methods have been proposed for them, leading to very efficient index compression schemes. This is important because query processing time is highly related to index access time. Thus, in this section, we also discuss some of the most important index compression techniques.

We first introduce basic concepts related to text compression. We then present some of the most important statistical compression methods, followed by a brief review of compression methods based on a dictionary. At the end, we discuss the application of compression to inverted files.

7.4.2 Basic Concepts

There are two general approaches to text compression: *statistical* and *dictionary* based. *Statistical methods* rely on generating good probability estimates (of appearance in the text) for each symbol. The more accurate the estimates are, the better the compression obtained. A *symbol* here is usually a character, a text word, or a fixed number of characters. The set of all possible symbols in the text is called the *alphabet*. The task of estimating the probability on each next symbol is called *modeling*. A *model* is essentially a collection of probability distributions, one for each context in which a symbol can be coded. Once these probabilities are available the symbols are converted into binary digits, a process called *coding*. In practice, both the encoder and decoder use the same model. The decoder interprets the output of the encoder (with reference to the same model) to find out the original symbol.

There are two well known statistical coding strategies: Huffman coding and arithmetic coding. The idea of Huffman coding is to assign a fixed-length bit encoding to each different symbol of the text. Compression is achieved by assigning a smaller number of bits to symbols with higher probabilities of appearance. Huffman coding was first proposed in the early 1950s and was the most important compression method until the late 1970s, when arithmetic coding made higher compression rates possible.

Arithmetic coding computes the code incrementally, one symbol at a time, as opposed to the Huffman coding scheme in which each different symbol is pre-encoded using a fixed-length number of bits. The incremental nature does not allow decoding a string which starts in the middle of a compressed file. To decode a symbol in the middle of a file compressed with arithmetic coding, it is necessary to decode the whole text from the very beginning until the desired word is reached. This characteristic makes arithmetic coding inadequate for use in an IR environment.

Dictionary methods substitute a sequence of symbols by a pointer to a previous occurrence of that sequence. The pointer representations are references to entries in a dictionary composed of a list of symbols (often called phrases) that are expected to occur frequently. Pointers to the dictionary entries are

chosen so that they need less space than the phrase they replace, thus obtaining compression. The distinction between modeling and coding does not exist in dictionary methods and there are no explicit probabilities associated to phrases. The most well known dictionary methods are represented by a family of methods, known as the Ziv-Lempel family.

Character-based Huffman methods are typically able to compress English texts to approximately five bits per character (usually, each uncompressed character takes 7-8 bits to be represented). More recently, a word-based Huffman method has been proposed as a better alternative for natural language texts. This method is able to reduce English texts to just over two bits per character. As we will see later on, word-based Huffman coding achieves compression rates close to the entropy and allows random access to intermediate points in the compressed text. Ziv-Lempel methods are able to reduce English texts to fewer than four bits per character. Methods based on arithmetic coding can also compress English texts to just over two bits per character. However, the price paid is slower compression and decompression, and the impossibility of randomly accessing intermediate points in the compressed text.

Before proceeding, let us present an important definition which will be useful from now on.

Definition Compression ratio *is the size of the compressed file as a fraction of the uncompressed file.*

7.4.3 Statistical Methods

In a statistical method, a probability is estimated for each symbol (the modeling task) and, based on this probability, a code is assigned to each symbol at a time (the coding task). Shorter codes are assigned to the most likely symbols.

The relationship between probabilities and codes was established by Claude Shannon in his source code theorem [718]. He showed that, in an optimal encoding scheme, a symbol that is expected to occur with probability p should be assigned a code of length $\log_2 \frac{1}{p}$ bits. The number of bits in which a symbol is best coded represents the *information content* of the symbol. The average amount of information per symbol over the whole alphabet is called the *entropy* of the probability distribution, and is given by:

$$E = \sum p_i \log_2 \frac{1}{p_i}$$

E is a *lower bound* on compression , measured in bits per symbol, which applies to any coding method based on the probability distribution p_i. It is important to note that E is calculated from the probabilities and so is a property of the model. See Chapter 6 for more details on this topic.

Modeling

The basic *function of a model* is to provide a probability assignment for the next symbol to be coded. High compression can be obtained by forming good models of the text that is to be coded. The probability assignment is explained in the following section.

Compression models can be *adaptive*, *static*, or *semi-static*. *Adaptive models* start with no information about the text and progressively learn about its statistical distribution as the compression process goes on. Thus, adaptive models need only one pass over the text and store no additional information apart from the compressed text. For long enough texts, such models converge to the true statistical distribution of the text. One major disadvantage, however, is that decompression of a file has to start from its beginning, since information on the distribution of the data is stored incrementally inside the file. Adaptive modeling is a good option for general purpose compression programs, but an inadequate alternative for full-text retrieval where random access to compressed patterns is a must. *Static models* assume an average distribution for all input texts. The modeling phase is done only once for all texts to be coded in the future (i.e., somehow a probability distribution is estimated and then used for all texts to be compressed in the future). These models tend to achieve poor compression ratios when the data deviates from initial statistical assumptions. For example, a model adequate for English literary texts will probably perform poorly for financial texts containing a lot of different numbers, as each number is relatively rare and so receives long codes.

Semi-static models do not assume any distribution on the data, but learn it in a first pass . In a second pass, they compress the data by using a fixed code derived from the distribution learned from the first pass. At decoding time, information on the data distribution is sent to the decoder before transmitting the encoded symbols. The disadvantages of semi-static models are that they must make two passes over the text and that information on the data distribution must be stored to be used by the decoder to decompress. In situations where interactive data communications are involved it may be impractical to make two passes over the text. However, semi-static models have a crucial advantage in IR contexts: since the same codes are used at every point in the compressed file, direct access is possible.

Word-based models take words instead of characters as symbols. Usually, a word is a contiguous string of characters in the set {A..Z, a..z} separated by other characters not in the set {A..Z, a..z}. There are many good reasons to use word-based models in an IR context. First, much better compression rates are achieved by taking words as symbols because words carry a lot of meaning in natural languages and, as a result, their distribution is much more related to the semantic structure of the text than the individual letters. Second, words are the atoms on which most information retrieval systems are built. Words are already stored for indexing purposes and so might be used as part of the model for compression. Third, the word frequencies are also useful in answering queries involving combinations of words because the best strategy is to start with the

least frequent words first.

Since the text is not only composed of words but also of separators, a model must also be chosen for them. There are many different ways to deal with separators. As words and separators always follow one another, two different alphabets are usually used: one for words and one for separators. Consider the following example: `each rose, a rose is a rose`. In the word-based model, the set of symbols of the alphabet is {`a`, `each`, `is`, `rose`}, whose frequencies are 2, 1, 1, and 3, respectively, and the set of separators is {',␣', ␣}, whose frequencies are 1 and 5, respectively (where ␣ represents a space). Once it is known that the text starts with a word or a separator, there is confusion about which alphabet to use.

In natural language texts, a word is followed by a single space in most cases. In the texts of the TREC-3 collection [342] (see Chapter 3), 70–80% of the separators are single spaces. Another good alternative is to consider the single space that follows a word as part of the same word. That is, if a word is followed by a space, we can encode just the word. If not, we can encode the word and then the following separator. At decoding time, we decode a word and assume that a space follows unless the next symbol corresponds to a separator. Notice that now a single alphabet for words and separators (single space excluded) is used. For instance, in the example above, the single alphabet is {',␣', `a`, `each`, `is`, `rose`} and there is no longer an alphabet for separators. As the alphabet excludes the single space then the words are called *spaceless words.*

In some situations word-based models for full-text databases have a potential to generate a great quantity of different codes and care must be exercised to deal with this fact. For instance, as discussed in the section on lexical analysis (at the beginning of this chapter), one has to consider whether a sequence of digits is to be considered as a word. If it is, then a collection which contains one million documents and includes document numbers as identifiers will generate one million words composed solely of digits, each one occurring once in the collection. This can be very inefficient for any kind of compression method available. One possible good solution is to divide long numbers into shorter ones by using a null (or implicit) punctuation marker in between. This diminishes the alphabet size resulting in considerable improvements in the compression ratio and in the decoding time.

Another important consideration is the size of the alphabet in word-based schemes. How large is the number of different words in a full-text database? It is empirically known that the vocabulary V of natural language texts with n words grows sublinearly. Heaps [352] shows that $V = O(n^{\beta})$, where β is a constant dependent on the particular text. For the 2 gigabyte TREC-3 collection [342], β is between 0.4 and 0.6 which means that the alphabet size grows roughly proportional to the square root of n. Even for this growth of the alphabet, the generalized Zipf law shows that the probability distribution is skewed so that the entropy remains constant. This implies that the compression ratio does not degrade as the text (and hence the number of different symbols) grows. Heaps' and Zipfs' laws are explained in Chapter 6.

Finally, it is important to mention that word-based Huffman methods need large texts to be effective (i.e., they are not adequate to compress and transmit

a single Web page over a network). The need to store the vocabulary represents an important space overhead when the text is small (say, less than 10 megabytes). However, this is not a concern in IR in general as the texts are large and the vocabulary is needed anyway for other purposes such as indexing and querying.

Coding

Coding corresponds to the task of obtaining the representation (code) of a symbol based on a probability distribution given by a model. The main goal of a coder is to assign short codes to likely symbols and long codes to unlikely ones. As we have seen in the previous section, the entropy of a probability distribution is a lower bound on how short the average length of a code can be, and the quality of a coder is measured in terms of how close to the entropy it is able to get. Another important consideration is the speed of both the coder and the decoder. Sometimes it is necessary to sacrifice the compression ratio to reduce the time to encode and decode the text.

A semi-static Huffman compression method works in two passes over the text. In a first pass, the modeler determines the probability distribution of the symbols and builds a coding tree according to this distribution. In a second pass, each next symbol is encoded according to the coding tree. Adaptive Huffman compression methods, instead, work in one single pass over the text updating the coding tree incrementally. The encoding of the symbols in the input text is also done during this single pass over the text. The main problem of adaptive Huffman methods is the cost of updating the coding tree as new symbols are read.

As with Huffman-based methods, arithmetic coding methods can also be based on static, semi-static or adaptive algorithms. The main strength of arithmetic coding methods is that they can generate codes which are arbitrarily close to the entropy for any kind of probability distribution. Another strength of arithmetic coding methods is that they do not need to store a coding tree explicitly. For adaptive algorithms, this implies that arithmetic coding uses less memory than Huffman-based coding. For static or semi-static algorithms, the use of canonical Huffman codes overcomes this memory problem (canonical Huffman trees are explained later on).

In arithmetic coding, the input text is represented by an interval of real numbers between 0 and 1. As the size of the input becomes larger, the interval becomes smaller and the number of bits needed to specify this interval increases. Compression is achieved because input symbols with higher probabilities reduce the interval less than symbols with smaller probabilities and hence add fewer bits to the output code.

Arithmetic coding presents many disadvantages over Huffman coding in an IR environment. First, arithmetic coding is much slower than Huffman coding, especially with static and semi-static algorithms. Second, with arithmetic coding, decompression cannot start in the middle of a compressed file. This contrasts with Huffman coding, in which it is possible to index and to decode from

any position in the compressed text if static or semi-static algorithms are used. Third, word-based Huffman coding methods yield compression ratios as good as arithmetic coding ones.

Consequently, Huffman coding is the method of choice in full-text retrieval, where both speed and random access are important. Thus, we will focus the remaining of our discussion on semi-static word-based Huffman coding.

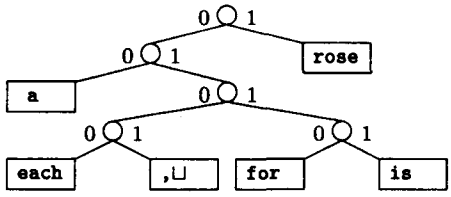
Huffman Coding

Huffman coding is one of the best known compression methods [386]. The idea is to assign a variable-length encoding in bits to each symbol and encode each symbol in turn. Compression is achieved by assigning shorter codes to more frequent symbols. Decompression uniqueness is guaranteed because no code is a prefix of another. A word-based semi-static model and Huffman coding form a good compression method for text.

Figure 7.2 presents an example of compression using Huffman coding on words. In this example the set of symbols of the alphabet is {'␣', a, each, for, is, rose}, whose frequencies are 1, 2, 1, 1, 1, and 3, respectively. In this case the alphabet is unique for words and separators. Notice that the separator '␣' is not part of the alphabet because the single space that follows a word is considered as part of the word. These words are called *spaceless words* (see more about spaceless words in Section 7.4.3). The Huffman tree shown in Figure 7.2 is an example of a binary trie built on binary codes. Tries are explained in Chapter 8.

Decompression is accomplished as follows. The stream of bits in the compressed file is traversed from left to right. The sequence of bits read is used to also traverse the Huffman compression tree, starting at the root. Whenever a leaf node is reached, the corresponding word (which constitutes the decompressed symbol) is printed out and the tree traversal is restarted. Thus, according to the tree in Figure 7.2, the presence of the code 0110 in the compressed file leads to the decompressed symbol **for**.

To build a Huffman tree, it is first necessary to obtain the symbols that constitute the alphabet and their probability distribution in the text to be compressed. The algorithm for building the tree then operates bottom up and starts



Original text: **for each rose, a rose is a rose**

Compressed text: 0110 0100 1 0101 00 1 0111 00 1

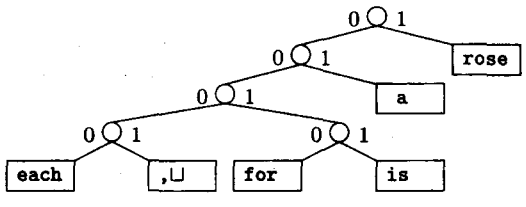
Figure 7.2 Huffman coding tree for spaceless words.

by creating for each symbol of the alphabet a node containing the symbol and its probability (or frequency). At this point there is a forest of one-node trees whose probabilities sum up to 1. Next, the two nodes with the smallest probabilities become children of a newly created parent node. With this parent node is associated a probability equal to the sum of the probabilities of the two chosen children. The operation is repeated ignoring nodes that are already children, until there is only one node, which becomes the root of the decoding tree. By delaying the pairing of nodes with high probabilities, the algorithm necessarily places them closer to the root node, making their code smaller. The two branches from every internal node are consistently labeled 0 and 1 (or 1 and 0). Given s symbols and their frequencies in the text, the algorithm builds the Huffman tree in $O(s \log s)$ time.

The number of Huffman trees which can be built for a given probability distribution is quite large. This happens because interchanging left and right subtrees of any internal node results in a different tree whenever the two subtrees are different in structure, but the weighted average code length is not affected. Instead of using any kind of tree, the preferred choice for most applications is to adopt a *canonical tree* which imposes a particular order to the coding bits.

A Huffman tree is canonical when the height of the left subtree of any node is never smaller than that of the right subtree, and all leaves are in increasing order of probabilities from left to right. Figure 7.3 shows the canonical tree for the example of Figure 7.2. The deepest leaf at the leftmost position of the Huffman canonical tree, corresponding to one element with smallest probability, will contain only zeros, and the following codes will be in increasing order inside each level. At each change of level we shift left one bit in the counting. The table in Figure 7.3 shows the canonical codes for the example of Figure 7.2.

A canonical code can be represented by an ordered sequence S of pairs (x_i, y_i), $1 \leq i \leq \ell$, where x_i represents the number of symbols at level i, y_i represents the numerical value of the first code at level i, and ℓ is the height of the tree. For our example in Figure 7.3, the ordered sequence is $S = \langle (1,1), (1,1), (0,\infty), (4,0) \rangle$. For instance, the fourth pair $(4,0)$ in S corresponds to the fourth level and indicates that there are four nodes at this level and that to the node most to the left is assigned a code, at this level, with value 0. Since this is the fourth level, a value 0 corresponds to the codeword 0000.



Symbol	Prob.	Old code	Can. code
each	1/9	0100	0000
,␣	1/9	0101	0001
for	1/9	0110	0010
is	1/9	0111	0011
a	2/9	00	01
rose	3/9	1	1

Figure 7.3 Canonical code.

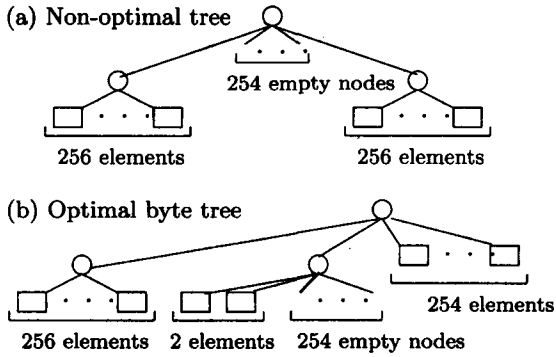


Figure 7.4 Example of byte Huffman tree.

One of the properties of canonical codes is that the set of codes having the same length are the binary representations of consecutive integers. Interpreted as integers, the 4-bit codes of the table in Figure 7.3 are 0, 1, 2, and 3, the 2-bit code is 1 and the 1-bit code is also 1. In our example, if the first character read from the input stream is 1, a codeword has been identified and the corresponding symbol can be output. If this value is 0, a second bit is appended and the two bits are again interpreted as an integer and used to index the table and identify the corresponding symbol. Once we read '00' we know that the code has four bits and therefore we can read two more bits and use them as an index into the table. This fact can be exploited to enable efficient encoding and decoding with small overhead. Moreover, much less memory is required, which is especially important for large vocabularies.

Byte-Oriented Huffman Code

The original method proposed by Huffman [386] leads naturally to binary coding trees. In [577], however, it is proposed to build the code assigned to each symbol as a sequence of whole bytes. As a result, the Huffman tree has degree 256 instead of 2. Typically, the code assigned to each symbol contains between 1 and 5 bytes. For example, a possible code for the word `rose` could be the 3-byte code '47 131 8.'

The construction of byte Huffman trees involves some details which must be dealt with. Care must be exercised to ensure that the first levels of the tree have no empty nodes when the code is not binary. Figure 7.4(a) illustrates a case where a naive extension of the binary Huffman tree construction algorithm might generate a non-optimal byte tree. In this example the alphabet has 512 symbols, all with the same probability. The root node has 254 empty spaces that could be occupied by symbols from the second level of the tree, changing their code lengths from 2 bytes to 1 byte.

A way to ensure that the empty nodes always go to the lowest level of the tree follows. We calculate beforehand the number of empty nodes that will arise.

We then compose these empty nodes with symbols of smallest probabilities (for moving the empty nodes to the deepest level of the final tree). To accomplish this, we need only to select a number of symbols equal to $1 + ((v - 256) \bmod 255)$, where v is the total number of symbols (i.e., the size of the vocabulary), for composing with the empty nodes. For instance, in the example in Figure 7.4(a), we have that 2 elements must be coupled with 254 empty nodes in the first step (because, $1 + ((512 - 256) \bmod 255) = 2$). The remaining steps are similar to the binary Huffman tree construction algorithm.

All techniques for efficient encoding and decoding mentioned previously can easily be extended to handle word-based byte Huffman coding. Moreover, no significant decrease of the compression ratio is experienced by using bytes instead of bits when the symbols are words. Further, decompression of byte Huffman code is faster than decompression of binary Huffman code. In fact, compression and decompression are very fast and compression ratios achieved are better than those of the Ziv-Lempel family [848, 849]. In practice byte processing is much faster than bit processing because bit shifts and masking operations are not necessary at decoding time or at searching time.

One important consequence of using byte Huffman coding is the possibility of performing direct searching on compressed text. The searching algorithm is explained in Chapter 8. The exact search can be done on the compressed text directly, using any known sequential pattern matching algorithm. Moreover, it allows a large number of variations of the exact and approximate compressed pattern matching problem, such as phrases, ranges, complements, wild cards, and arbitrary regular expressions. The algorithm is based on a word-oriented shift-or algorithm and on a fast Boyer-Moore-type filter. For approximate searching on the compressed text it is eight times faster than an equivalent approximate searching on the uncompressed text, thanks to the use of the vocabulary by the algorithm [577, 576]. This technique is not only useful in speeding up sequential search. It can also be used to improve indexed schemes that combine inverted files and sequential search, like Glimpse [540].

7.4.4 Dictionary Methods

Dictionary methods achieve compression by replacing groups of consecutive symbols (or phrases) with a pointer to an entry in a dictionary. Thus, the central decision in the design of a dictionary method is the selection of entries in the dictionary. The choice of phrases can be made by static, semi-adaptive, or adaptive algorithms. The simplest dictionary schemes use static dictionaries containing short phrases. Static dictionary encoders are fast as they demand little effort for achieving a small amount of compression. One example that has been proposed several times in different forms is the digram coding, where selected pairs of letters are replaced with codewords. At each step the next two characters are inspected and verified if they correspond to a digram in the dictionary. If so, they are coded together and the coding position is shifted by two characters; otherwise, the single character is represented by its normal code and the coding

position is shifted by one character.

The main problem with static dictionary encoders is that the dictionary might be suitable for one text and unsuitable for another. One way to avoid this problem is to use a semi-static dictionary scheme, constructing a new dictionary for each text to be compressed. However, the problem of deciding which phrases should be put in the dictionary is not an easy task at all. One elegant solution to this problem is to use an adaptive dictionary scheme, such as the one proposed in the 1970s by Ziv and Lempel.

The Ziv-Lempel type of adaptive dictionary scheme uses the idea of replacing strings of characters with a reference to a previous occurrence of the string. This approach is effective because most characters can be coded as part of a string that has occurred earlier in the text. If the pointer to an earlier occurrence of a string is stored in fewer bits than the string it replaces then compression is achieved.

Adaptive dictionary methods present some disadvantages over the statistical word-based Huffman method. First, they do not allow decoding to start in the middle of a compressed file. As a consequence direct access to a position in the compressed text is not possible, unless the entire text is decoded from the beginning until the desired position is reached. Second, dictionary schemes are still popular for their speed and economy of memory, but the new results in statistical methods make them the method of choice in an IR environment. Moreover, the improvement of computing technology will soon make statistical methods feasible for general use, and the interest in dictionary methods will eventually decrease.

7.4.5 Inverted File Compression

As already discussed, an inverted file is typically composed of (a) a vector containing all the distinct words in the text collection (which is called the *vocabulary*) and (b) for each word in the vocabulary, a list of all documents in which that word occurs. Inverted files are widely used to index large text files. The size of an inverted file can be reduced by compressing the inverted lists. Because the list of document numbers within the inverted list is in ascending order, it can also be considered as a sequence of *gaps* between document numbers. Since processing is usually done sequentially starting from the beginning of the list, the original document numbers can always be recomputed through sums of the gaps.

By observing that these gaps are small for frequent words and large for infrequent words, compression can be obtained by encoding small values with shorter codes. One possible coding scheme for this case is the *unary code*, in which an integer x is coded as $(x-1)$ one bits followed by a zero bit, so the code for the integer 3 is 110. The second column of Table 7.1 shows unary codes for integers between 1 and 10.

Elias [235] presented two other variable-length coding schemes for integers. One is Elias-γ code, which represents the number x by a concatenation of two

Gap x	Unary	Elias-γ	Elias-δ	Golomb $b = 3$
1	0	0	0	00
2	10	100	1000	010
3	110	101	1001	011
4	1110	11000	10100	100
5	11110	11001	10101	1010
6	111110	11010	10110	1011
7	1111110	11011	10111	1100
8	11111110	1110000	11000000	11010
9	111111110	1110001	11000001	11011
10	1111111110	1110010	11000010	11100

Table 7.1 Example codes for integers.

parts: (1) a unary code for $1+\lfloor \log x \rfloor$ and (2) a code of $\lfloor \log x \rfloor$ bits that represents the value of $x - 2^{\lfloor \log x \rfloor}$ in binary. For $x = 5$, we have that $1 + \lfloor \log x \rfloor = 3$ and that $x - 2^{\lfloor \log x \rfloor} = 1$. Thus, the Elias-$\gamma$ code for $x = 5$ is generated by combining the unary code for 3 (code 110) with the 2-bits binary number for 1 (code 01) which yields the codeword 11001. Other examples of Elias-γ codes are shown in Table 7.1.

The other coding scheme introduced by Elias is the Elias-δ code, which represents the prefix indicating the number of binary bits by the Elias-γ code rather than the unary code. For $x = 5$, the first part is then 101 instead of 110. Thus, the Elias-δ codeword for $x = 5$ is 10101. In general, the Elias-δ code for an arbitrary integer x requires $1 + 2\lfloor \log \log 2x \rfloor + \lfloor \log x \rfloor$ bits. Table 7.1 shows other examples of Elias-δ codes. In general, for small values of x the Elias-γ codes are shorter than the Elias-δ codes. However, in the limit, as x becomes large, the situation is reversed.

Golomb [307] presented another run-length coding method for positive integers. The Golomb code is very effective when the probability distribution is geometric. With inverted files, the likelihood of a gap being of size x can be computed as the probability of having $x - 1$ non-occurrences (within consecutively numbered documents) of that particular word followed by one occurrence. If a word occurs within a document with a probability p, the probability of a gap of size x is then

$$Pr[x] = (1 - p)^{x-1} p$$

which is the *geometric distribution.* In this case, the model is parameterized and makes use of the actual density of pointers in the inverted file. Let N be the number of documents in the system and V be the size of the vocabulary. Then, the probability p that any randomly selected document contains any randomly

chosen term can be estimated as

$$p = \frac{number\ of\ pointers}{N \times V}$$

where the number of pointers represent the 'size' of the index.

The Golomb method works as follows. For some parameter b, a gap $x > 0$ is coded as $q+1$ in unary, where $q = \lfloor (x-1)/b \rfloor$, followed by $r = (x-1) - q \times b$ coded in binary, requiring either $\lfloor \log b \rfloor$ or $\lceil \log b \rceil$ bits. That is, if $r < 2^{\lfloor \log b \rfloor - 1}$ then the number coded in binary requires $\lfloor \log b \rfloor$ bits, otherwise it requires $\lceil \log b \rceil$ bits where the first bit is 1 and the remaining bits assume the value $r - 2^{\lfloor \log b \rfloor - 1}$ coded in $\lfloor \log b \rfloor$ binary digits. For example, with $b = 3$ there are three possible remainders, and those are coded as 0, 10, and 11, for $r = 0$, $r = 1$, and $r = 2$, respectively. Similarly, for $b = 5$ there are five possible remainders r, 0 through 4, and these are assigned codes 00, 01, 100, 101, and 110. Then, if the value $x = 9$ is to be coded relative to $b = 3$, calculation yields $q = 2$ and $r = 2$, because $9-1 = 2\times 3+2$. Thus, the encoding is 110 followed by 11. Relative to $b = 5$, the values calculated are $q = 1$ and $r = 1$, resulting in a code of 10 followed by 101.

To operate with the Golomb compression method, it is first necessary to establish the parameter b for each term. For gap compression, an appropriate value is $b \approx 0.69(N/f_t)$, where N is the total number of documents and f_t is the number of documents that contain term t. Witten, Moffat and Bell [825] present a detailed study of different text collections. For all of their practical work on compression of inverted lists, they use Golomb code for the list of gaps. In this case Golomb code gives better compression than either Elias-γ or Elias-δ. However, it has the disadvantage of requiring two passes to be generated, since it requires knowledge of f_t, the number of documents containing term t.

Moffat and Bell [572] show that the index for the 2 gigabytes TREC-3 collection, which contains 162,187,989 pointers and 894,406 distinct terms, when coded with Golomb code, occupies 132 megabytes. Considering the average number of bits per pointer, they obtained 5.73, 6.19, and 6.43 using Golomb, Elias-δ, and Elias-γ, respectively.

7.5 Comparing Text Compression Techniques

Table 7.2 presents a comparison between arithmetic coding, character-based Huffman coding, word-based Huffman coding, and Ziv-Lempel coding, considering the aspects of compression ratio, compression speed, decompression speed, memory space overhead, compressed pattern matching capability, and random access capability.

One important objective of any compression method is to be able to obtain good compression ratios. It seems that two bits per character (or 25% compression ratio) is a very good result for natural language texts. Thus, 'very good' in the context of Table 7.2 means a compression ratio under 30%, 'good' means a compression ratio between 30% and 45%, and 'poor' means a compression ratio over 45%.

	Arithmetic	Character Huffman	Word Huffman	Ziv-Lempel
Compression ratio	very good	poor	very good	good
Compression speed	slow	fast	fast	very fast
Decompression speed	slow	fast	very fast	very fast
Memory space	low	low	high	moderate
Compressed pat. matching	no	yes	yes	yes
Random access	no	yes	yes	no

Table 7.2 Comparison of the main techniques.

Two other important characteristics of a compression method are compression and decompression speeds. Measuring the speed of various compression methods is difficult because it depends on the implementation details of each method, the compiler used, the computer architecture of the machine used to run the program, and so on. Considering compression speed, the LZ78 methods (Unix *compress* is an example) are among the fastest. Considering decompression speed, the LZ77 methods (*gzip* is an example) from the Ziv-Lempel are among the fastest.

For statistical methods (e.g., arithmetic and semi-static Huffman) the compression time includes the cost of the first pass during which the probability distribution of the symbols are obtained. With two passes over the text to compress, the Huffman-based methods are slower than some Ziv-Lempel methods, but not very far behind. On the other hand, arithmetic methods are slower than Huffman methods because of the complexity of arithmetic coding compared with canonical Huffman coding. Considering decompression speed, word-based Huffman methods are as fast as Ziv-Lempel methods, while character-based Huffman methods are slower than word-based Huffman methods. Again, the complexity of arithmetic coding make them slower than Huffman coding during decompression.

All Ziv-Lempel compression methods require a moderate amount of memory during encoding and decoding to store tables containing previously occurring strings. In general, more detailed tables that require more memory for storage yield better compression. Statistical methods store the probability distribution of the symbols of the text during the modeling phase, and the model during both compression and decompression phases. Consequently, the amount of memory depends on the size of the vocabulary of the text in each case, which is high for word-based models and low for character-based models.

In an IR environment, two important considerations are whether the compression method allows efficient random access and direct searching on compressed text (or compressed pattern matching). Huffman methods allow random access and decompression can start anywhere in the middle of a compressed file, while arithmetic coding and Ziv-Lempel methods cannot. More recently, practical, efficient, and flexible direct searching methods on compressed texts have been discovered for word-based Huffman compression [575, 576, 577].

Direct searching has also been proposed for Ziv-Lempel methods, but only on a theoretical basis, with no implementation of the algorithms [250, 19].

More recently, Navarro and Raffinot [592] presented some preliminary implementations of algorithms to search directly Ziv-Lempel compressed text. Their algorithms are twice as fast as decompressing and searching, but slower than searching the decompressed text. They are also able to extract data from the middle of the compressed text without necessarily decompressing everything, and although some previous text has to be decompressed (i.e., it is not really 'direct access'), the amount of work is proportional to the size of the text to be decompressed (and not to its position in the compressed text).

7.6 Trends and Research Issues

In this chapter we covered various text transformation techniques which we call simply text operations. We first discussed five distinct text operations for preprocessing a document text and generating a set of index terms for searching and querying purposes. These five text operations were here called lexical analysis, elimination of stopwords, stemming, selection of index terms, and thesauri. The first four are directly related to the generation of a good set of index terms. The fifth, construction of a thesaurus, is more related to the building of categorization hierarchies which are used for capturing term relationships. These relationships can then be used for expanding the user query (manually or automatically) towards a formulation which better suits the user information need.

Nowadays, there is controversy regarding the potential improvements to retrieval performance generated by stopwords elimination, stemming, and index terms selection. In fact, there is no conclusive evidence that such text operations yield consistent improvements in retrieval performance. As a result, modern retrieval systems might not use these text operations at all. A good example of this trend is the fact that some Web search engines index all the words in the text regardless of their syntactic nature or their role in the text.

Furthermore, it is also not clear that automatic query expansion using thesaurus-based techniques can yield improved retrieval performance. The same cannot be said of the use of a thesaurus to directly assist the user with the query formation process. In fact, the success of the 'Yahoo!' Web search engine, which uses a term categorization hierarchy to show term relationships to the user, is an indication that thesaurus-based techniques might be quite useful with the highly interactive interfaces being developed for modern digital library systems.

We also briefly discussed the operation of clustering. Since clustering is more an operation of grouping documents than an operation of text transformation, we did not cover it thoroughly here. For a more complete coverage of clustering the reader is referred to Chapter 5.

One text operation rather distinct from the previous ones is compression. While the previous text operations aim, in one form or another, at improving the quality of the answer set, the operation of compressing text aims at reducing space, I/O, communication costs, and searching faster in the compressed text (exactly or approximately). In fact, the gain obtained from compressing text is

that it requires less storage space, takes less time to be transmitted, and permits efficient direct and sequential access to compressed text.

For effective operation in an IR environment, a compression method should satisfy the following requirements: good compression ratio, fast coding, fast decoding, fast random access without the need to decode from the beginning, and direct searching without the need to decompress the compressed text. A good compression ratio saves space in secondary storage and reduces communication costs. Fast coding reduces processing overhead due to the introduction of compression into the system. Sometimes, fast decoding is more important than fast coding, as in documentation systems in which a document is compressed once and decompressed many times from disk. Fast random access allows efficient processing of multiple queries submitted by the users of the information system. We compared various compression schemes using these requirements as parameters. We have seen that it is much faster to search sequentially a text compressed by a word-based byte Huffman encoding scheme than to search the uncompressed version of the text. Our discussion suggests that word-based byte Huffman compression (which has been introduced only very recently) shows great promise as an effective compression scheme for modern information retrieval systems.

We also discussed the application of compression to index structures such as inverted files. Inverted files are composed of several inverted lists which are themselves formed by document numbers organized in ascending order. By coding the difference between these document numbers, efficient compression can be attained.

The main trends in text compression today are the use of semi-static word-based modeling and Huffman coding. The new results in statistical methods, such as byte-Huffman coding, suggest that they are preferable methods for use in an IR environment. Further, with the possibility now of directly searching the compressed text, and the recent work [790] of Vo and Moffat on efficient manipulation of compressed indices, the trend is towards maintaining both the index and the text compressed at all times, unless the user wants to visualize the uncompressed text.

7.7 Bibliographic Discussion

Our discussion on lexical analysis and elimination of stopwords is based on the work of Fox [263]. For stemming, we based our discussion on the work of Frakes [274]. The Porter stemming algorithm detailed in the appendix is from [648], while our coverage of thesauri is based on the work of Foskett [261]. Here, however, we did not cover automatic generation of thesauri. Such discussion can be found in Chapter 5 and in [739, 735]. Additional discussion on the usefulness of thesauri is presented in [419, 735].

Regarding text compression, several books are available. Most of the topics discussed here are covered in more detail by Witten, Moffat and Bell [825]. They also present implementations of text compression methods, such as Huffman and arithmetic coding, as part of a fully operational retrieval system written in ANSI

C. Bell, Cleary and Witten [78] cover statistical and dictionary methods, laying particular stress on adaptive methods as well as theoretical aspects of compression, with estimates on the entropy of several natural languages. Storer [747] covers the main compression techniques, with emphasis on dictionary methods.

Huffman coding was originally presented in [386]. Adaptive versions of Huffman coding appear in [291, 446, 789]. Word-based compression is considered in [81, 571, 377, 77]. Bounds on the inefficiency of Huffman coding have been presented by [291]. Canonical codes were first presented in [713]. Many properties of the canonical codes are mentioned in [374]. Byte Huffman coding was proposed in [577]. Sequential searching on byte Huffman compressed text is described in [577, 576].

Sequential searching on Ziv-Lempel compressed data is presented in [250, 19]. More recently, implementations of sequential searching on Ziv-Lempel compressed text are presented in [593]. One of the first papers on arithmetic coding is in [675]. Other references are [823, 78].

A variety of compression methods for inverted lists are studied in [573]. The most effective compression methods for inverted lists are based on the sequence of gaps between document numbers, as considered in [77] and in [572]. Their results are based on run-length encodings proposed by Elias [235] and Golomb [307]. A comprehensive study of inverted file compression can be found in [825]. More recently Vo and Moffat [790] have presented algorithms to process the index with no need to fully decode the compressed index.

Chapter 8
Indexing and Searching

with Gonzalo Navarro

8.1 Introduction

Chapter 4 describes the query operations that can be performed on text databases. In this chapter we cover the main techniques we need to implement those query operations.

We first concentrate on searching queries composed of words and on reporting the documents where they are found. The number of occurrences of a query in each document and even its exact positions in the text may also be required. Following that, we concentrate on algorithms dealing with Boolean operations. We then consider sequential search algorithms and pattern matching. Finally, we consider structured text and compression techniques.

An obvious option in searching for a basic query is to scan the text sequentially. Sequential or online text searching involves finding the occurrences of a pattern in a text when the text is not preprocessed. Online searching is appropriate when the text is small (i.e., a few megabytes), and it is the only choice if the text collection is very volatile (i.e., undergoes modifications very frequently) or the index space overhead cannot be afforded.

A second option is to build data structures over the text (called *indices*) to speed up the search. It is worthwhile building and maintaining an index when the text collection is large and *semi-static*. Semi-static collections can be updated at reasonably regular intervals (e.g., daily) but they are not deemed to support thousands of insertions of single words per second, say. This is the case for most real text databases, not only dictionaries or other slow growing literary works. For instance, it is the case for Web search engines or journal archives.

Nowadays, the most successful techniques for medium size databases (say up to 200Mb) combine online and indexed searching.

We cover three main indexing techniques: inverted files, suffix arrays, and signature files. Keyword-based search is discussed first. We emphasize inverted files, which are currently the best choice for most applications. Suffix trees

and arrays are faster for phrase searches and other less common queries, but are harder to build and maintain. Finally, signature files were popular in the 1980s, but nowadays inverted files outperform them. For all the structures we pay attention not only to their search cost and space overhead, but also to the cost of building and updating them.

We assume that the reader is familiar with basic data structures, such as sorted arrays, binary search trees, B-trees, hash tables, and tries. Since tries are heavily used we give a brief and simplified reminder here. Tries, or digital search trees, are multiway trees that store sets of strings and are able to retrieve any string in time proportional to its length (independent of the number of strings stored). A special character is added to the end of the string to ensure that no string is a prefix of another. Every edge of the tree is labeled with a letter. To search a string in a trie, one starts at the root and scans the string character-wise, descending by the appropriate edge of the trie. This continues until a leaf is found (which represents the searched string) or the appropriate edge to follow does not exist at some point (i.e., the string is not in the set). See Figure 8.3 for an example of a text and a trie built on its words.

Although an index must be built prior to searching it, we present these tasks in the reverse order. We think that understanding first how a data structure is used makes it clear how it is organized, and therefore eases the understanding of the construction algorithm, which is usually more complex.

Throughout this chapter we make the following assumptions. We call n the size of the text database. Whenever a pattern is searched, we assume that it is of length m, which is much smaller than n. We call M the amount of main memory available. We assume that the modifications which a text database undergoes are additions, deletions, and replacements (which are normally made by a deletion plus an addition) of pieces of text of size $n' < n$.

We give experimental measures for many algorithms to give the reader a grasp of the real times involved. To do this we use a reference architecture throughout the chapter, which is representative of the power of today's computers. We use a 32-bit Sun UltraSparc-1 of 167 MHz with 64 Mb of RAM, running Solaris. The code is written in C and compiled with all optimization options. For the text data, we use collections from TREC-2, specifically WSJ, DOE, FR, ZIFF and AP. These are described in more detail in Chapter 3.

8.2 Inverted Files

An inverted file (or inverted index) is a word-oriented mechanism for indexing a text collection in order to speed up the searching task. The inverted file structure is composed of two elements: the *vocabulary* and the *occurrences*. The vocabulary is the set of all different words in the text. For each such word a list of all the text positions where the word appears is stored. The set of all those lists is called the 'occurrences' (Figure 8.1 shows an example). These positions can refer to words or characters. Word positions (i.e., position i refers to the i-th word) simplify

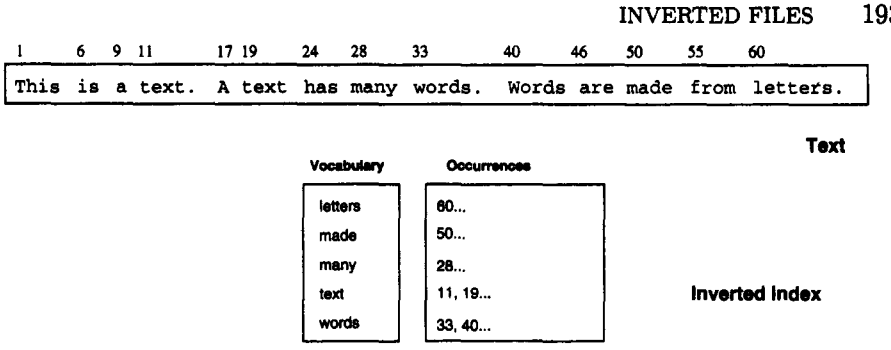


Figure 8.1 A sample text and an inverted index built on it. The words are converted to lower-case and some are not indexed. The occurrences point to character positions in the text.

phrase and proximity queries, while character positions (i.e., the position i is the i-th character) facilitate direct access to the matching text positions.

Some authors make the distinction between inverted files and inverted lists. In an inverted file, each element of a list points to a document or file name, while inverted lists match our definition. We prefer not to make such a distinction because, as we will see later, this is a matter of the *addressing granularity*, which can range from text positions to logical blocks.

The space required for the vocabulary is rather small. According to Heaps' law (see Chapter 6) the vocabulary grows as $O(n^\beta)$, where β is a constant between 0 and 1 dependent on the text, being between 0.4 and 0.6 in practice. For instance, for 1 Gb of the TREC-2 collection the vocabulary has a size of only 5 Mb. This may be further reduced by stemming and other normalization techniques as described in Chapter 7.

The occurrences demand much more space. Since each word appearing in the text is referenced once in that structure, the extra space is $O(n)$. Even omitting stopwords (which is the default practice when words are indexed), in practice the space overhead of the occurrences is between 30% and 40% of the text size.

To reduce space requirements, a technique called *block addressing* is used. The text is divided in blocks, and the occurrences point to the blocks where the word appears (instead of the exact positions). The classical indices which point to the exact occurrences are called 'full inverted indices.' By using block addressing not only can the pointers be smaller because there are fewer blocks than positions, but also all the occurrences of a word inside a single block are collapsed to one reference (see Figure 8.2). Indices of only 5% overhead over the text size are obtained with this technique. The price to pay is that, if the exact occurrence positions are required (for instance, for a proximity query), then an online search over the qualifying blocks has to be performed. For instance, block addressing indices with 256 blocks stop working well with texts of 200 Mb.

Table 8.1 presents the projected space taken by inverted indices for texts of

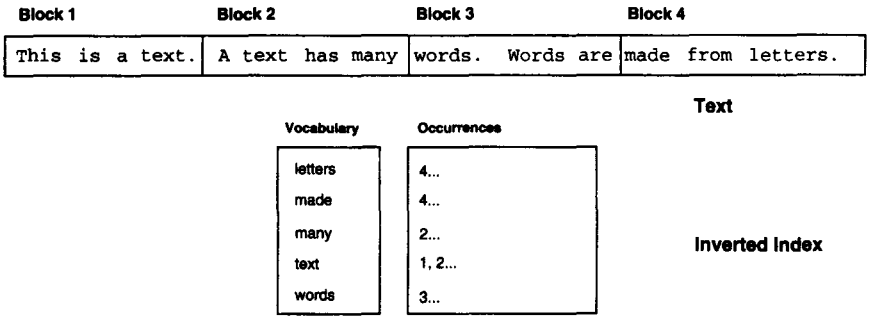


Figure 8.2 The sample text split into four blocks, and an inverted index using block addressing built on it. The occurrences denote block numbers. Notice that both occurrences of '`words`' collapsed into one.

different sizes, with and without the use of stopwords. The full inversion stands for inverting all the words and storing their exact positions, using four bytes per pointer. The document addressing index assumes that we point to documents which are of size 10 Kb (and the necessary number of bytes per pointer, i.e. one, two, and three bytes, depending on text size). The block addressing index assumes that we use 256 or 64K blocks (one or two bytes per pointer) independently of the text size. The space taken by the pointers can be significantly reduced by using compression. We assume that 45% of all the words are stopwords, and that there is one non-stopword each 11.5 characters. Our estimation for the vocabulary is based on Heaps' law with parameters $V = 30n^{0.5}$. All these decisions were taken according to our experience and experimentally validated.

The blocks can be of fixed size (imposing a logical block structure over the text database) or they can be defined using the natural division of the text collection into files, documents, Web pages, or others. The division into blocks of fixed size improves efficiency at retrieval time, i.e. the more variance in the block sizes, the more amount of text sequentially traversed on average. This is because larger blocks match queries more frequently and are more expensive to traverse.

Alternatively, the division using natural cuts may eliminate the need for online traversal. For example, if one block per retrieval unit is used and the exact match positions are not required, there is no need to traverse the text for single-word queries, since it is enough to know which retrieval units to report. But if, on the other hand, many retrieval units are packed into a single block, the block has to be traversed to determine which units to retrieve.

It is important to notice that in order to use block addressing, the text must be readily available at search time. This is not the case for remote text (as in Web search engines), or if the text is in a CD-ROM that has to be mounted, for instance. Some restricted queries not needing exact positions can still be solved if the blocks are retrieval units.

Index	Small collection (1 Mb)		Medium collection (200 Mb)		Large collection (2 Gb)	
Addressing words	45%	73%	36%	64%	35%	63%
Addressing documents	19%	26%	18%	32%	26%	47%
Addressing 64K blocks	27%	41%	18%	32%	5%	9%
Addressing 256 blocks	18%	25%	1.7%	2.4%	0.5%	0.7%

Table 8.1 Sizes of an inverted file as approximate percentages of the size the whole text collection. Four granularities and three collections are considered. For each collection, the right column considers that stopwords are not indexed while the left column considers that all words are indexed.

8.2.1 Searching

The search algorithm on an inverted index follows three general steps (some may be absent for specific queries):

- **Vocabulary search** The words and patterns present in the query are isolated and searched in the vocabulary. Notice that phrases and proximity queries are split into single words.
- **Retrieval of occurrences** The lists of the occurrences of all the words found are retrieved.
- **Manipulation of occurrences** The occurrences are processed to solve phrases, proximity, or Boolean operations. If block addressing is used it may be necessary to directly search the text to find the information missing from the occurrences (e.g., exact word positions to form phrases).

Hence, searching on an inverted index always starts in the vocabulary. Because of this it is a good idea to have it in a separate file. It is possible that this file fits in main memory even for large text collections.

Single-word queries can be searched using any suitable data structure to speed up the search, such as hashing, tries, or B-trees. The first two give $O(m)$ search cost (independent of the text size). However, simply storing the words in lexicographical order is cheaper in space and very competitive in performance, since the word can be binary searched at $O(\log n)$ cost. Prefix and range queries can also be solved with binary search, tries, or B-trees, but not with hashing. If the query is formed by single words, then the process ends by delivering the list of occurrences (we may need to make a union of many lists if the pattern matches many words).

Context queries are more difficult to solve with inverted indices. Each element must be searched separately and a list (in increasing positional order) generated for each one. Then, the lists of all elements are traversed in synchronization to find places where all the words appear in sequence (for a phrase) or appear close enough (for proximity). If one list is much shorter than the others, it may be better to binary search its elements into the longer lists instead of performing a linear merge. It is possible to prove using Zipf's law that this is normally the case. This is important because the most time-demanding operation on inverted indices is the merging or intersection of the lists of occurrences.

If the index stores character positions the phrase query cannot allow the separators to be disregarded, and the proximity has to be defined in terms of character distance.

Finally, note that if block addressing is used it is necessary to traverse the blocks for these queries, since the position information is needed. It is then better to intersect the lists to obtain the blocks which contain all the searched words and then sequentially search the context query in those blocks as explained in section 8.5. Some care has to be exercised at block boundaries, since they can split a match. This part of the search, if present, is also quite time consuming.

Using Heaps' and the generalized Zipf's laws, it has been demonstrated that the cost of solving queries is sublinear in the text size, even for complex queries involving list merging. The time complexity is $O(n^{\alpha})$, where α depends on the query and is close to 0.4..0.8 for queries with reasonable selectivity.

Even if block addressing is used and the blocks have to be traversed, it is possible to select the block size as an increasing function of n, so that not only does the space requirement keep sublinear but also the amount of text traversed in all useful queries is also sublinear.

Practical figures show, for instance, that both the space requirement and the amount of text traversed can be close to $O(n^{0.85})$. Hence, inverted indices allow us to have sublinear search time at sublinear space requirements. This is not possible on the other indices.

Search times on our reference machine for a full inverted index built on 250 Mb of text give the following results: searching a simple word took 0.08 seconds, while searching a phrase took 0.25 to 0.35 seconds (from two to five words).

8.2.2 Construction

Building and maintaining an inverted index is a relatively low cost task. In principle, an inverted index on a text of n characters can be built in $O(n)$ time. All the vocabulary known up to now is kept in a trie data structure, storing for each word a list of its occurrences (text positions). Each word of the text is read and searched in the trie. If it is not found, it is added to the trie with an empty list of occurrences. Once it is in the trie, the new position is added to the end of its list of occurrences. Figure 8.3 illustrates this process.

Once the text is exhausted, the trie is written to disk together with the list of occurrences. It is good practice to split the index into two files. In the

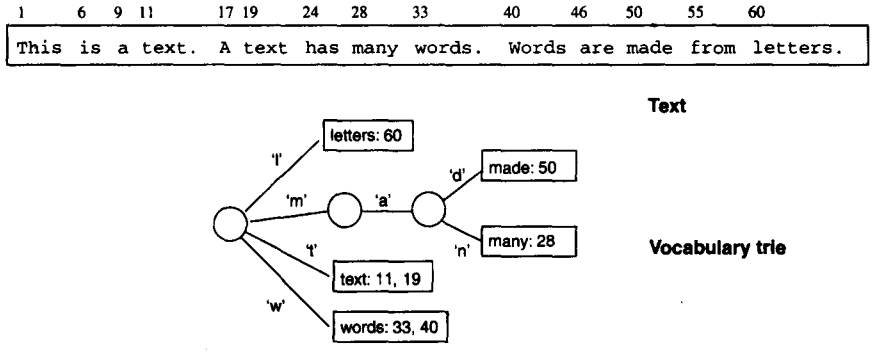


Figure 8.3 Building an inverted index for the sample text.

first file, the lists of occurrences are stored contiguously. In this scheme, the file is typically called a 'posting file'. In the second file, the vocabulary is stored in lexicographical order and, for each word, a pointer to its list in the first file is also included. This allows the vocabulary to be kept in memory at search time in many cases. Further, the number of occurrences of a word can be immediately known from the vocabulary with little or no space overhead.

We analyze now the construction time under this scheme. Since in the trie $O(1)$ operations are performed per text character, and the positions can be inserted at the end of the lists of occurrences in $O(1)$ time, the overall process is $O(n)$ worst-case time.

However, the above algorithm is not practical for large texts where the index does not fit in main memory. A paging mechanism will severely degrade the performance of the algorithm. We describe an alternative which is faster in practice.

The algorithm already described is used until the main memory is exhausted (if the trie takes up too much space it can be replaced by a hash table or other structure). When no more memory is available, the *partial* index I_i obtained up to now is written to disk and erased from main memory before continuing with the rest of the text.

Finally, a number of partial indices I_i exist on disk. These indices are then merged in a hierarchical fashion. Indices I_1 and I_2 are merged to obtain the index $I_{1..2}$; I_3 and I_4 produce $I_{3..4}$; and so on. The resulting partial indices are now approximately twice the size. When all the indices at this level have been merged in this way, the merging proceeds at the next level, joining the index $I_{1..2}$ with the index $I_{3..4}$ to form $I_{1..4}$. This is continued until there is just one index comprising the whole text, as illustrated in Figure 8.4.

Merging two indices consists of merging the sorted vocabularies, and whenever the same word appears in both indices, merging both lists of occurrences. By construction, the occurrences of the smaller-numbered index are before those of the larger-numbered index, and therefore the lists are just concatenated. This is a very fast process in practice, and its complexity is $O(n_1 + n_2)$, where n_1 and n_2 are the sizes of the indices.

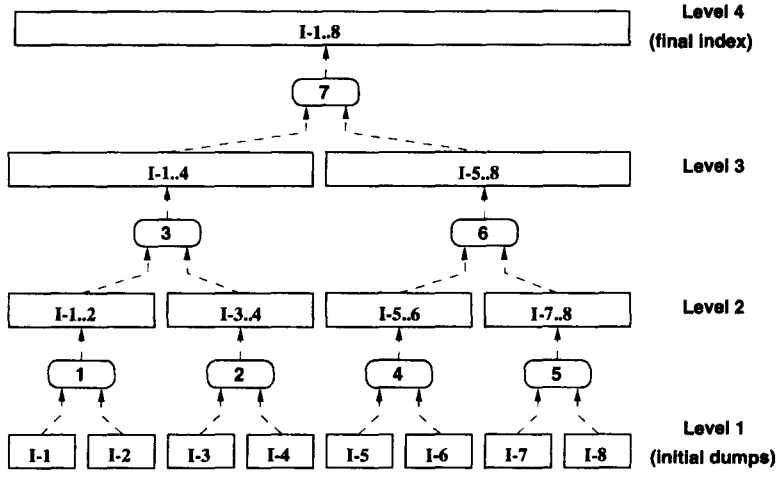


Figure 8.4 Merging the partial indices in a binary fashion. Rectangles represent partial indices, while rounded rectangles represent merging operations. The numbers inside the merging operations show a possible merging order.

The total time to generate the partial indices is $O(n)$ as before. The number of partial indices is $O(n/M)$. Each level of merging performs a linear process over the whole index (no matter how it is split into partial indices at this level) and thus its cost is $O(n)$. To merge the $O(n/M)$ partial indices, $\log_2(n/M)$ merging levels are necessary, and therefore the cost of this algorithm is $O(n \log(n/M))$.

More than two indices can be merged at once. Although this does not change the complexity, it improves efficiency since fewer merging levels exist. On the other hand, the memory buffers for each partial index to merge will be smaller and hence more disk seeks will be performed. In practice it is a good idea to merge even 20 partial indices at once.

Real times to build inverted indices on the reference machine are between 4-8 Mb/min for collections of up to 1 Gb (the slowdown factor as the text grows is barely noticeable). Of this time, 20-30% is spent on merging the partial indices.

To reduce build-time space requirements, it is possible to perform the merging in-place. That is, when two or more indices are merged, write the result in the same disk blocks of the original indices instead of on a new file. It is also a good idea to perform the hierarchical merging as soon as the files are generated (e.g., collapse I_1 and I_2 into $I_{1..2}$ as soon as I_2 is produced). This also reduces space requirements because the vocabularies are merged and redundant words are eliminated (there is no redundancy in the occurrences). The vocabulary can be a significative part of the smaller partial indices, since they represent a small text.

This algorithm changes very little if block addressing is used. Index maintenance is also cheap. Assume that a new text of size n' is added to the database. The inverted index for the new text is built and then both indices are merged

as is done for partial indices. This takes $O(n + n' \log(n'/M))$. Deleting text can be done by an $O(n)$ pass over the index eliminating the occurrences that point inside eliminated text areas (and eliminating words if their lists of occurrences disappear in the process).

8.3 Other Indices for Text

8.3.1 Suffix Trees and Suffix Arrays

Inverted indices assume that the text can be seen as a sequence of words. This restricts somewhat the kinds of queries that can be answered. Other queries such as phrases are expensive to solve. Moreover, the concept of word does not exist in some applications such as genetic databases.

In this section we present suffix arrays. Suffix arrays are a space efficient implementation of suffix trees. This type of index allows us to answer efficiently more complex queries. Its main drawbacks are its costly construction process, that the text must be readily available at query time, and that the results are not delivered in text position order. This structure can be used to index only words (without stopwords) as the inverted index as well as to index any text character. This makes it suitable for a wider spectrum of applications, such as genetic databases. However, for word-based applications, inverted files perform better unless complex queries are an important issue.

This index sees the text as one long string. Each position in the text is considered as a text *suffix* (i.e., a string that goes from that text position to the end of the text). It is not difficult to see that two suffixes starting at different positions are lexicographically different (assume that a character smaller than all the rest is placed at the end of the text). Each suffix is thus uniquely identified by its position.

Not all text positions need to be indexed. *Index points* are selected from the text, which point to the *beginning* of the text positions which will be retrievable. For instance, it is possible to index only word beginnings to have a functionality similar to inverted indices. Those elements which are not index points are not retrievable (as in an inverted index it is not possible to retrieve the middle of a word). Figure 8.5 illustrates this.

Structure

In essence, a suffix tree is a trie data structure built over all the suffixes of the text. The pointers to the suffixes are stored at the leaf nodes. To improve space utilization, this trie is compacted into a Patricia tree. This involves compressing unary paths, i.e. paths where each node has just one child. An indication of the next character position to consider is stored at the nodes which root a compressed path. Once unary paths are not present the tree has $O(n)$ nodes instead of the worst-case $O(n^2)$ of the trie (see Figure 8.6).

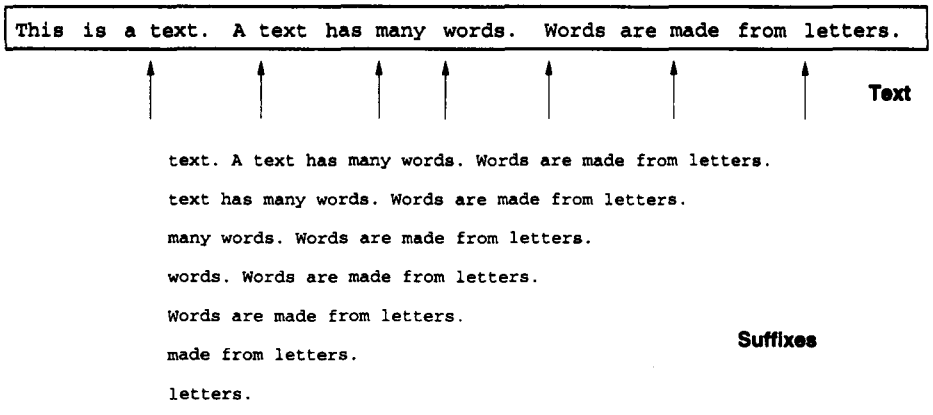


Figure 8.5 The sample text with the index points of interest marked. Below, the suffixes corresponding to those index points.

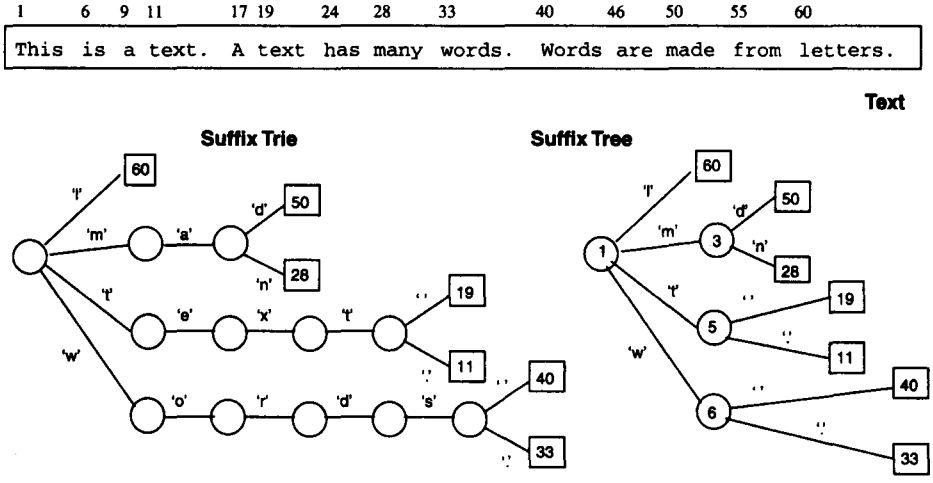


Figure 8.6 The suffix trie and suffix tree for the sample text.

The problem with this structure is its space. Depending on the implementation, each node of the trie takes 12 to 24 bytes, and therefore even if only word beginnings are indexed, a space overhead of 120% to 240% over the text size is produced.

Suffix arrays provide essentially the same functionality as suffix trees with much less space requirements. If the leaves of the suffix tree are traversed in left-to-right order (top to bottom in our figures), all the suffixes of the text are retrieved in lexicographical order. A suffix array is simply an array containing all the pointers to the text suffixes listed in lexicographical order, as shown in Figure 8.7. Since they store one pointer per indexed suffix, the space requirements

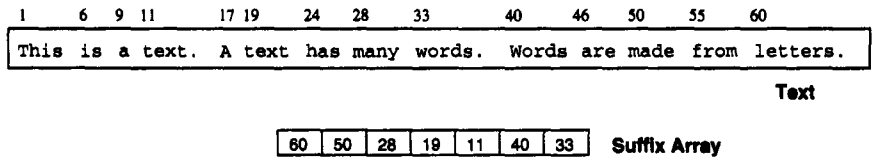


Figure 8.7 The suffix array for the sample text.

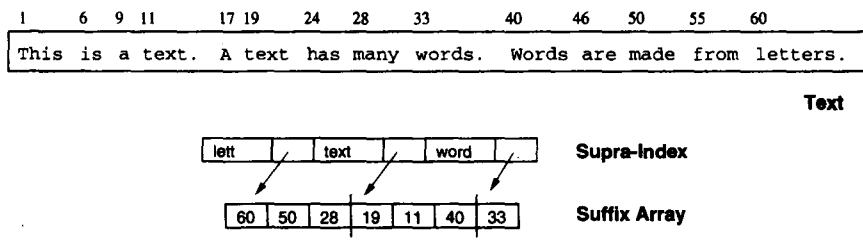


Figure 8.8 A supra-index over our suffix array. One out of three entries are sampled, keeping their first four characters. The pointers (arrows) are in fact unnecessary.

are almost the same as those for inverted indices (disregarding compression techniques), i.e. close to 40% overhead over the text size.

Suffix arrays are designed to allow binary searches done by comparing the contents of each pointer. If the suffix array is large (the usual case), this binary search can perform poorly because of the number of random disk accesses. To remedy this situation, the use of *supra-indices* over the suffix array has been proposed. The simplest supra-index is no more than a sampling of one out of b suffix array entries, where for each sample the first ℓ suffix characters are stored in the supra-index. This supra-index is then used as a first step of the search to reduce external accesses. Figure 8.8 shows an example.

This supra-index does not in fact need to take samples at fixed intervals, nor to take samples of the same length. For word-indexing suffix arrays it has been suggested that a new sample could be taken each time the first word of the suffix changes, and to store the word instead of ℓ characters. This is exactly the same as having a vocabulary of the text plus pointers to the array. In fact, the only important difference between this structure and an inverted index is that the occurrences of each word in an inverted index are sorted by text position, while in a suffix array they are sorted lexicographically by the text following the word. Figure 8.9 illustrates this relationship.

The extra space requirements of supra-indices are modest. In particular, it is clear that the space requirements of the suffix array with a vocabulary supra-index are exactly the same as for inverted indices (except for compression, as we see later).

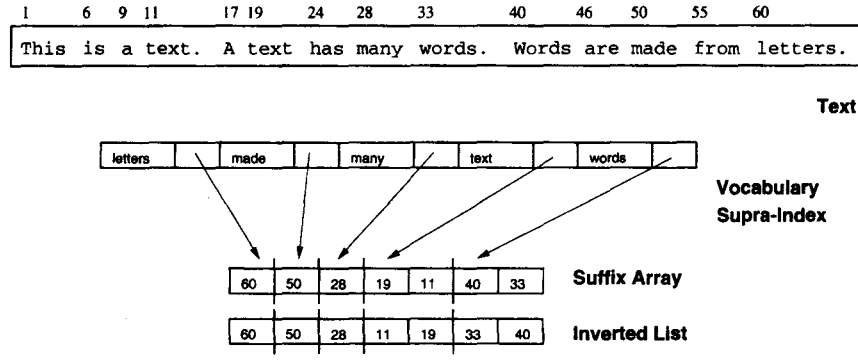


Figure 8.9 Relationship between our inverted list and suffix array with vocabulary supra-index.

Searching

If a suffix tree on the text can be afforded, many basic patterns such as words, prefixes, and phrases can be searched in $O(m)$ time by a simple trie search. However, suffix trees are not practical for large texts, as explained. Suffix arrays, on the other hand, can perform the same search operations in $O(\log n)$ time by doing a binary search instead of a trie search.

This is achieved as follows: the search pattern originates two 'limiting patterns' P_1 and P_2, so that we want any suffix S such that $P_1 \leq S < P_2$. We binary search both limiting patterns in the suffix array. Then, all the elements lying between both positions point to exactly those suffixes that start like the original pattern (i.e., to the pattern positions in the text). For instance, in our example of figure 8.9, in order to find the word 'text' we search for 'text' and 'texu', obtaining the portion of the array that contains the pointers 19 and 11.

All these queries retrieve a subtree of the suffix tree or an interval of the suffix array. The results have to be collected later, which may imply sorting them in ascending text order. This is a complication of suffix trees or arrays with respect to inverted indices.

Simple phrase searching is a good case for these indices. A simple phrase of words can be searched as if it was a simple pattern. This is because the suffix tree/array sorts with respect to the complete suffixes and not only their first word. A proximity search, on the other hand, has to be solved element-wise. The matches for each element must be collected and sorted and then they have to be intersected as for inverted files.

The binary search performed on suffix arrays, unfortunately, is done on disk, where the accesses to (random) text positions force a seek operation which spans the disk tracks containing the text. Since a random seek is $O(n)$ in size, this makes the search cost $O(n \log n)$ time. Supra-indices are used as a first step in any binary search operation to alleviate this problem. To avoid performing $O(\log n)$ random accesses to the text on disk (and to the suffix array on disk), the search starts in the supra-index, which usually fits in main memory (text samples

included). After this search is completed, the suffix array block which is between the two selected samples is brought into memory and the binary search is completed (performing random accesses to the text on disk). This reduces disk search times to close to 25% of the original time. Modified binary search techniques that sacrifice the exact partition in the middle of the array taking into account the current disk head position allow a further reduction from 40% to 60%.

Search times in a 250 Mb text in our reference machine are close to 1 second for a simple word or phrase, while the part corresponding to the accesses to the text sums up 0.6 seconds. The use of supra-indices should put the total time close to 0.3 seconds. Note that the times, although high for simple words, do not degrade for long phrases as with inverted indices.

Construction in Main Memory

A suffix tree for a text of n characters can be built in $O(n)$ time. The algorithm, however, performs poorly if the suffix tree does not fit in main memory, which is especially stringent because of the large space requirements of the suffix trees. We do not cover the linear algorithm here because it is quite complex and only of theoretical interest.

We concentrate on direct suffix array construction. Since the suffix array is no more than the set of pointers lexicographically sorted, the pointers are collected in ascending text order and then just sorted by the text they point to. Note that in order to compare two suffix array entries the corresponding text positions must be accessed. These accesses are basically random. Hence, both the suffix array and the text must be in main memory. This algorithm costs $O(n \log n)$ string comparisons.

An algorithm to build the suffix array in $O(n \log n)$ character comparisons follows. All the suffixes are bucket-sorted in $O(n)$ time according to the first letter only. Then, each bucket is bucket-sorted again, now according to their first two letters. At iteration i, the suffixes begin already sorted by their 2^{i-1} first letters and end up sorted by their first 2^i letters. As at each iteration the total cost of all the bucket sorts is $O(n)$, the total time is $O(n \log n)$, and the average is $O(n \log \log n)$ (since $O(\log n)$ comparisons are necessary on average to distinguish two suffixes of a text). This algorithm accesses the text only in the first stage (bucket sort for the first letter).

In order to sort the strings in the i-th iteration, notice that since *all* suffixes are sorted by their first 2^{i-1} letters, to sort the text positions $T_{a...}$ and $T_{b...}$ in the suffix array (assuming that they are in the same bucket, i.e., they share their first 2^{i-1} letters), it is enough to determine the relative order between text positions $T_{a+2^{i-1}}$ and $T_{b+2^{i-1}}$ in the current stage of the search. This can be done in constant time by storing the reverse permutation. We do not enter here into further detail.

Construction of Suffix Arrays for Large Texts

There is still the problem that large text databases will not fit in main memory. It could be possible to apply an external memory sorting algorithm. However,

each comparison involves accessing the text at random positions on the disk. This will severely degrade the performance of the sorting process.

We explain an algorithm especially designed for large texts. Split the text into blocks that can be sorted in main memory. Then, for each block, build its suffix array in main memory and merge it with the rest of the array already built for the previous text. That is:

- build the suffix array for the first block,
- build the suffix array for the second block,
- merge both suffix arrays,
- build the suffix array for the third block,
- merge the new suffix array with the previous one,
- build the suffix array for the fourth block,
- merge the new suffix array with the previous one,
- ... and so on.

The difficult part is how to merge a large suffix array (already built) with the small suffix array (just built). The merge needs to compare text positions which are spread in a large text, so the problem persists. The solution is to first determine how many elements of the large array are to be placed between each pair of elements in the small array, and later use that information to merge the arrays without accessing the text. Hence, the information that we need is how many suffixes of the large text lie between each pair of positions of the small suffix array. We compute counters that store this information.

The counters are computed without using the large suffix array. The text corresponding to the large array is sequentially read into main memory. Each suffix of that text is *searched* in the small suffix array (in main memory). Once we find the inter-element position where the suffix lies, we just increment the appropriate counter. Figure 8.10 illustrates this process.

We analyze this algorithm now. If there is $O(M)$ main memory to index, then there will be $O(n/M)$ text blocks. Each block is merged against an array of size $O(n)$, where all the $O(n)$ suffixes of the large text are binary searched in the small suffix array. This gives a total CPU complexity of $O(n^2 \log(M)/M)$.

Notice that this same algorithm can be used for index maintenance. If a new text of size n' is added to the database, it can be split into blocks as before and merged block-wise into the current suffix array. This will take $O(nn' \log(M)/M)$. To delete some text it suffices to perform an $O(n)$ pass over the array eliminating all the text positions which lie in the deleted areas.

As can be seen, the construction process is in practice more costly for suffix arrays than for inverted files. The construction of the supra-index consists of a fast final sequential pass over the suffix array.

Indexing times for 250 Mb of text are close to 0.8 Mb/min on the reference machine. This is five to ten times slower than the construction of inverted indices.

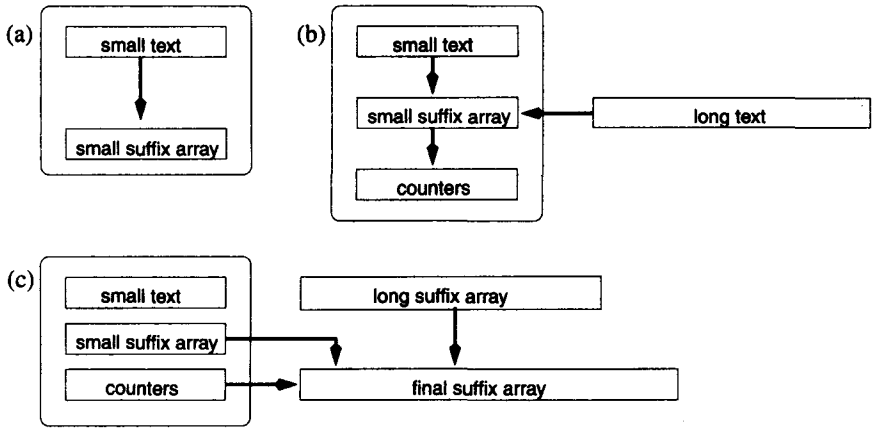


Figure 8.10 A step of the suffix array construction for large texts: (*a*) the local suffix array is built, (*b*) the counters are computed, (*c*) the suffix arrays are merged.

8.3.2 Signature Files

Signature files are word-oriented index structures based on hashing. They pose a low overhead (10% to 20% over the text size), at the cost of forcing a sequential search over the index. However, although their search complexity is linear (instead of sublinear as with the previous approaches), its constant is rather low, which makes the technique suitable for not very large texts. Nevertheless, inverted files outperform signature files for most applications.

Structure

A signature file uses a hash function (or 'signature') that maps words to bit masks of B bits. It divides the text in blocks of b words each. To each text block of size b, a bit mask of size B will be assigned. This mask is obtained by bitwise ORing the signatures of all the words in the text block. Hence, the signature file is no more than the sequence of bit masks of all blocks (plus a pointer to each block). The main idea is that if a word is present in a text block, then all the bits set in its signature are also set in the bit mask of the text block. Hence, whenever a bit is set in the mask of the query word and not in the mask of the text block, then the word is not present in the text block. Figure 8.11 shows an example.

However, it is possible that all the corresponding bits are set even though the word is not there. This is called a *false drop*. The most delicate part of the design of a signature file is to ensure that the probability of a false drop is low enough while keeping the signature file as short as possible.

The hash function is forced to deliver bit masks which have at least ℓ bits set. A good model assumes that ℓ bits are randomly set in the mask (with possible repetition). Let $\alpha = \ell/B$. Since each of the b words sets ℓ bits at

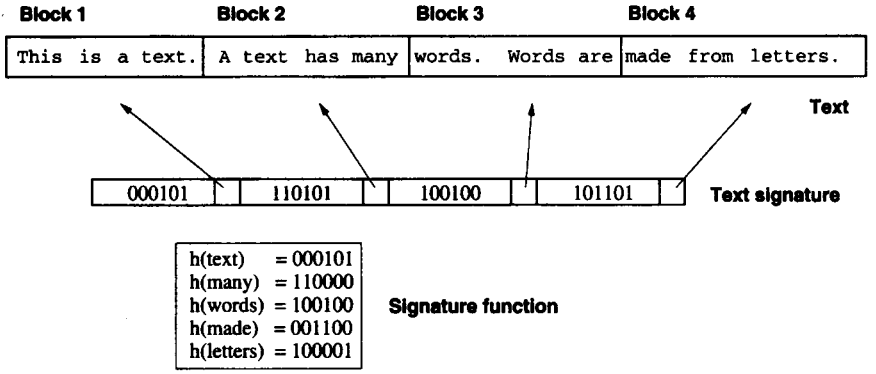


Figure 8.11 A signature file for our sample text cut into blocks.

random, the probability that a given bit of the mask is set in a word signature is $1-(1-1/B)^{b\ell} \approx 1-e^{-b\alpha}$. Hence, the probability that the ℓ random bits set in the query are also set in the mask of the text block is

$$(1-e^{-b\alpha})^{\alpha B}$$

which is minimized for $\alpha = \ln(2)/b$. The false drop probability under the optimal selection $\ell = B\ln(2)/b$ is $(1/2^{\ln(2)})^{B/b} = 1/2^{\ell}$.

Hence, a reasonable proportion B/b must be determined. The space overhead of the index is approximately $(1/80) \times (B/b)$ because B is measured in bits and b in words. Then, the false drop probability is a function of the overhead to pay. For instance, a 10% overhead implies a false drop probability close to 2%, while a 20% overhead errs with probability 0.046%. This error probability corresponds to the expected amount of sequential searching to perform while checking if a match is a false drop or not.

Searching

Searching a single word is carried out by hashing it to a bit mask W, and then comparing the bit masks B_i of all the text blocks. Whenever $(W \ \& \ B_i = W)$, where & is the bitwise AND, all the bits set in W are also set in B_i and therefore the text block *may* contain the word. Hence, for all candidate text blocks, an online traversal must be performed to verify if the word is actually there. This traversal cannot be avoided as in inverted files (except if the risk of a false drop is accepted).

No other types of patterns can be searched in this scheme. On the other hand, the scheme is more efficient to search phrases and reasonable proximity queries. This is because *all* the words must be present in a block in order for that block to hold the phrase or the proximity query. Hence, the bitwise OR of all the query masks is searched, so that *all* their bits must be present. This

reduces the probability of false drops. This is the only indexing scheme which improves in phrase searching.

Some care has to be exercised at block boundaries, however, to avoid missing a phrase which crosses a block limit. To allow searching phrases of j words or proximities of up to j words, consecutive blocks must overlap in j words.

If the blocks correspond to retrieval units, simple Boolean conjunctions involving words or phrases can also be improved by forcing all the relevant words to be in the block.

We were only able to find real performance estimates from 1992, run on a Sun 3/50 with local disk. Queries on a small 2.8 Mb database took 0.42 seconds. Extrapolating to today's technology, we find that the performance should be close to 20 Mb/sec (recall that it is linear time), and hence the example of 250 Mb of text would take 12 seconds, which is quite slow.

Construction

The construction of a signature file is rather easy. The text is simply cut in blocks, and for each block an entry of the signature file is generated. This entry is the bitwise OR of the signatures of all the words in the block.

Adding text is also easy, since it is only necessary to keep adding records to the signature file. Text deletion is carried out by deleting the appropriate bit masks.

Other storage proposals exist apart from storing all the bit masks in sequence. For instance, it is possible to make a different file for each bit of the mask, i.e. one file holding all the first bits, another file for all the second bits, etc. This reduces the disk times to search for a query, since only the files corresponding to the ℓ bits which are set in the query have to be traversed.

8.4 Boolean Queries

We now cover set manipulation algorithms. These algorithms are used when operating on sets of results, which is the case in Boolean queries. Boolean queries are described in Chapter 4, where the concept of *query syntax tree* is defined.

Once the leaves of the query syntax tree are solved (using the algorithms to find the documents containing the basic queries given), the relevant documents must be worked on by composition operators. Normally the search proceeds in three phases: the first phase determines which documents classify, the second determines the relevance of the classifying documents so as to present them appropriately to the user, and the final phase retrieves the exact positions of the matches to highlight them in those documents that the user actually wants to see.

This scheme avoids doing unnecessary work on documents which will not classify at last (first phase), or will not be read at last (second phase). However, some phases can be merged if doing the extra operations is not expensive. Some phases may not be present at all in some scenarios.

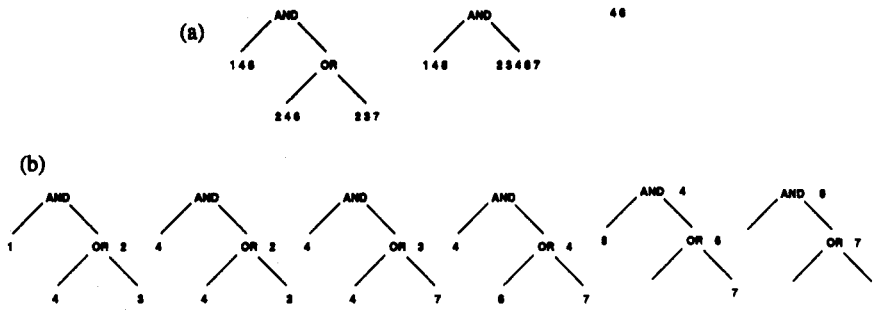


Figure 8.12 Processing the internal nodes of the query syntax tree. In (*a*) full evaluation is used. In (*b*) we show lazy evaluation in more detail.

Once the leaves of the query syntax tree find the classifying sets of documents, these sets are further operated by the internal nodes of the tree. It is possible to algebraically optimize the tree using identities such as $a\ OR\ (a\ AND\ b) = a$, for instance, or sharing common subexpressions, but we do not cover this issue here.

As all operations need to pair the same document in both their operands, it is good practice to keep the sets sorted, so that operations like intersection, union, etc. can proceed sequentially on both lists and also generate a sorted list. Other representations for sets not consisting of the list of matching documents (such as bit vectors) are also possible.

Under this scheme, it is possible to evaluate the syntax tree in *full* or *lazy* form. In the full evaluation form, both operands are first completely obtained and then the complete result is generated. In lazy evaluation, results are delivered only when required, and to obtain that result some data is recursively required to both operands.

Full evaluation allows some optimizations to be performed because the sizes of the results are known in advance (for instance, merging a very short list against a very long one can proceed by binary searching the elements of the short list in the long one). Lazy evaluation, on the other hand, allows the application to control when to do the work of obtaining new results, instead of blocking it for a long time. Hybrid schemes are possible, for example obtain all the leaves at once and then proceed in lazy form. This may be useful, for instance, to implement some optimizations or to ensure that all the accesses to the index are sequential (thus reducing disk seek times). Figure 8.12 illustrates this.

The complexity of solving these types of queries, apart from the cost of obtaining the results at the leaves, is normally linear in the total size of all the intermediate results. This is why this time may dominate the others, when there are huge intermediate results. This is more noticeable to the user when the final result is small.

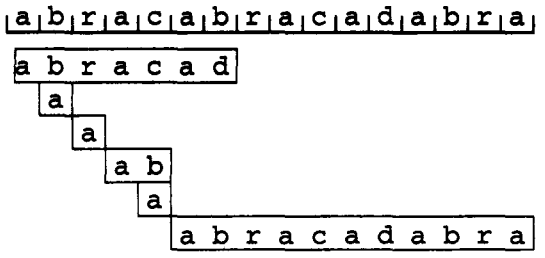


Figure 8.13 Brute-force search algorithm for the pattern '**abracadabra.**' Squared areas show the comparisons performed.

8.5 Sequential Searching

We now cover the algorithms for text searching when no data structure has been built on the text. As shown, this is a basic part of some indexing techniques as well as the only option in some cases. We cover exact string matching in this section. Later we cover matching of more complex patterns. Our exposition is mainly conceptual and the implementation details are not shown (see the Bibliographic Discussion at the end of this chapter for more information).

The problem of exact string matching is: given a short pattern P of length m and a long text T of length n, find all the text positions where the pattern occurs. With minimal changes this problem subsumes many basic queries, such as word, prefix, suffix, and substring search.

This is a classical problem for which a wealth of solutions exists. We sketch the main algorithms, and leave aside a lot of the theoretical work that is not competitive in practice. For example, we do not include the Karp-Rabin algorithm, which is a nice application of hashing to string searching, but is not practical. We also briefly cover multipattern algorithms (that search many patterns at once), since a query may have many patterns and it may be more efficient to retrieve them all at once. Finally, we also mention how to do phrases and proximity searches.

We assume that the text and the pattern are sequences of characters drawn from an alphabet of size σ, whose first character is at position 1. The average-case analysis assumes random text and patterns.

8.5.1 Brute Force

The brute-force (BF) algorithm is the simplest possible one. It consists of merely trying all possible pattern positions in the text. For each such position, it verifies whether the pattern matches at that position. See Figure 8.13.

Since there are $O(n)$ text positions and each one is examined at $O(m)$ worst-case cost, the worst-case of brute-force searching is $O(mn)$. However, its average

case is $O(n)$ (since on random text a mismatch is found after $O(1)$ comparisons on average). This algorithm does not need any pattern preprocessing.

Many algorithms use a modification of this scheme. There is a *window* of length m which is slid over the text. It is *checked* whether the text in the window is equal to the pattern (if it is, the window position is reported as a match). Then, the window is *shifted* forward. The algorithms mainly differ in the way they check and shift the window.

8.5.2 Knuth-Morris-Pratt

The KMP algorithm was the first with linear worst-case behavior, although on average it is not much faster than BF. This algorithm also slides a window over the text. However, it does not try all window positions as BF does. Instead, it reuses information from previous checks.

After the window is checked, whether it matched the pattern or not, a number of pattern letters were compared to the text window, and they all matched except possibly the last one compared. Hence, when the window has to be shifted, there is a *prefix* of the pattern that matched the text. The algorithm takes advantage of this information to avoid trying window positions which can be deduced not to match.

The pattern is preprocessed in $O(m)$ time and space to build a table called *next*. The *next* table at position j says which is the longest proper prefix of $P_{1..j-1}$ which is also a suffix and the characters following prefix and suffix are different. Hence $j - next[j] + 1$ window positions can be safely skipped if the characters up to $j-1$ matched, and the j-th did not. For instance, when searching the word '`abracadabra`,' if a text window matched up to '`abracab`,' five positions can be safely skipped since $next[7] = 1$. Figure 8.14 shows an example.

The crucial observation is that this information depends only on the pattern, because if the text in the window matched up to position $j-1$, then that text is equal to the pattern.

The algorithm moves a window over the text and a pointer inside the window. Each time a character matches, the pointer is advanced (a match is reported if the pointer reaches the end of the window). Each time a character is not matched, the window is shifted forward in the text, to the position given by *next*, but the pointer position in the text does not change. Since at each text comparison the window or the pointer advance by at least one position, the algorithm performs at most $2n$ comparisons (and at least n).

The Aho-Corasick algorithm can be regarded as an extension of KMP in matching a set of patterns. The patterns are arranged in a trie-like data structure. Each trie node represents having matched a prefix of some pattern(s). The *next* function is replaced by a more general set of *failure* transitions. Those transitions go between nodes of the trie. A transition leaving from a node representing the prefix x leads to a node representing a prefix y, such that y is the longest prefix in the set of patterns which is also a proper suffix of x. Figure 8.15 illustrates this.

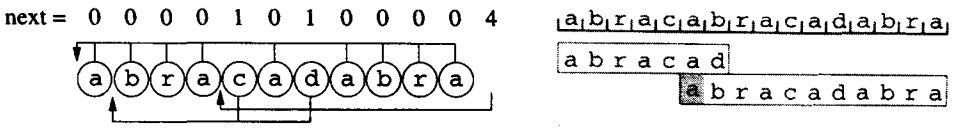


Figure 8.14 KMP algorithm searching '**abracadabra**.' On the left, an illustration of the *next* function. Notice that after matching '**abracada**' we do not try to match the last '**a**' with the first one since what follows cannot be a '**b**.' On the right, a search example. Grayed areas show the prefix information reused.

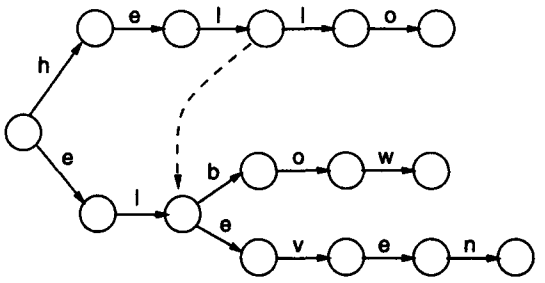


Figure 8.15 Aho-Corasick trie example for the set '**hello**,' '**elbow**' and '**eleven**' showing only one of all the failure transitions.

This trie, together with its failure transitions, is built in $O(m)$ time and space (where m is the total length of all the patterns). Its search time is $O(n)$ no matter how many patterns are searched. Much as KMP, it makes at most $2n$ inspections.

8.5.3 Boyer-Moore Family

BM algorithms are based on the fact that the check inside the window can proceed backwards. When a match or mismatch is determined, a *suffix* of the pattern has been compared and found equal to the text in the window. This can be used in a way very similar to the *next* table of KMP, i.e. compute for every pattern position j the next-to-last occurrence of $P_{j..m}$ inside P. This is called the 'match heuristic.'

This is combined with what is called the 'occurrence heuristic.' It states that the text character that produced the mismatch (if a mismatch occurred) has to be aligned with the same character in the pattern after the shift. The heuristic which gives the longest shift is selected.

For instance, assume that '**abracadabra**' is searched in a text which starts with '**abracababra**.' After matching the suffix '**abra**' the underlined text character '**b**' will cause a mismatch. The match heuristic states that since '**abra**' was matched a shift of 7 is safe. The occurrence heuristic states that since the underlined '**b**' must match the pattern, a shift of 5 is safe. Hence, the pattern is

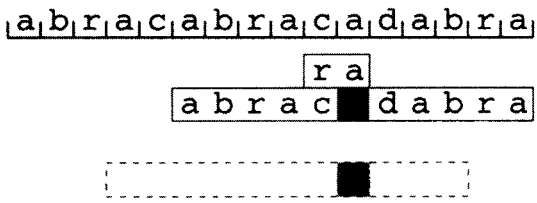


Figure 8.16 BM algorithm searching 'abracadabra.' Squared areas show the comparisons performed. Grayed areas have already been compared (but the algorithm compares them again). The dashed box shows the match heuristic, which was not chosen.

shifted by 7. See Figure 8.16.

The preprocessing time and space of this algorithm is $O(m+\sigma)$. Its search time is $O(n\log(m)/m)$ on average, which is 'sublinear' in the sense that not all characters are inspected. On the other hand, its worst case is $O(mn)$ (unlike KMP, the old suffix information is not kept to avoid further comparisons).

Further simplifications of the BM algorithm lead to some of the fastest algorithms on average. The Simplified BM algorithm uses only the occurrence heuristic. This obtains almost the same shifts in practice. The BM-Horspool (BMH) algorithm does the same, but it notices that it is not important any more that the check proceeds backwards, and uses the occurrence heuristic on the *last* character of the window instead of the one that caused the mismatch. This gives longer shifts on average. Finally, the BM-Sunday (BMS) algorithm modifies BMH by using the character *following* the last one, which improves the shift especially on short patterns.

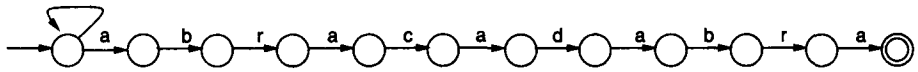
The Commentz-Walter algorithm is an extension of BM to multipattern search. It builds a trie on the reversed patterns, and instead of a backward window check, it enters into the trie with the window characters read backwards. A shift function is computed by a natural extension of BM. In general this algorithm improves over Aho-Corasick for not too many patterns.

8.5.4 Shift-Or

Shift-Or is based on *bit-parallelism*. This technique involves taking advantage of the intrinsic parallelism of the bit operations inside a computer word (of w bits). By cleverly using this fact, the number of operations that an algorithm performs can be cut by a factor of at most w. Since in current architectures w is 32 or 64, the speedup is very significant in practice.

The Shift-Or algorithm uses bit-parallelism to simulate the operation of a non-deterministic automaton that searches the pattern in the text (see Figure 8.17). As this automaton is simulated in time $O(mn)$, the Shift-Or algorithm achieves $O(mn/w)$ worst-case time (optimal speedup).

The algorithm first builds a table B which for each character stores a bit mask $b_m...b_1$. The mask in $B[c]$ has the i-th bit set to zero if and only if



$B[a]$ =	1	0	0	1	0	1	0	1	0	0	1
$B[b]$ =	0	1	0	0	0	0	0	0	1	0	0
$B[r]$ =	0	0	1	0	0	0	0	0	0	1	0
$B[c]$ =	0	0	0	0	1	0	0	0	0	0	0
$B[d]$ =	0	0	0	0	0	0	1	0	0	0	0
$B[*]$ =	0	0	0	0	0	0	0	0	0	0	0

Figure 8.17 Non-deterministic automaton that searches '**abracadabra**,' and the associated B table. The initial self-loop matches any character. Each table column corresponds to an edge of the automaton.

$p_i = c$ (see Figure 8.17). The state of the search is kept in a machine word $D = d_m...d_1$, where d_i is zero whenever the state numbered i in Figure 8.17 is active. Therefore, a match is reported whenever d_m is zero. In the following, we use '|' to denote the bitwise OR and '&' to denote the bitwise AND.

D is set to all ones originally, and for each new text character T_j, D is updated using the formula

$$D' \leftarrow (D << 1) \mid B[T_j]$$

(where '<<' means shifting all the bits in D one position to the left and setting the rightmost bit to zero). It is not hard to relate the formula to the movement that occurs in the non-deterministic automaton for each new text character.

For patterns longer than the computer word (i.e., $m > w$), the algorithm uses $\lceil m/w \rceil$ computer words for the simulation (not all them are active all the time). The algorithm is $O(n)$ on average and the preprocessing is $O(m+\sigma)$ time and $O(\sigma)$ space.

It is easy to extend Shift-Or to handle classes of characters by manipulating the B table and keeping the search algorithm unchanged.

This paradigm also can search a large set of extended patterns, as well as multiple patterns (where the complexity is the same as before if we consider that m is the total length of all the patterns).

8.5.5 Suffix Automaton

The Backward DAWG matching (BDM) algorithm is based on a suffix automaton. A *suffix automaton* on a pattern P is an automaton that recognizes all the suffixes of P. The non-deterministic version of this automaton has a very regular structure and is shown in Figure 8.18.

The BDM algorithm converts this automaton to deterministic. The size and construction time of this automaton is $O(m)$. This is basically the preprocessing effort of the algorithm. Each path from the initial node to any internal

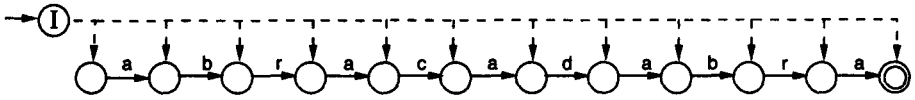


Figure 8.18 A non-deterministic suffix automaton. Dashed lines represent ε-transitions (i.e., they occur without consuming any input). I is the initial state of the automaton.

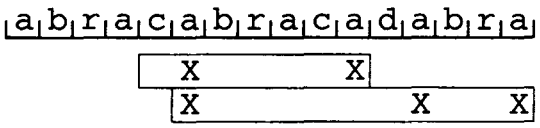


Figure 8.19 The BDM algorithm for the pattern '**abracadabra.**' The rectangles represent elements compared to the text window. The **X**s show the positions where a pattern prefix was recognized.

node represents a substring of the pattern. The final nodes represent pattern suffixes.

To search a pattern P, the suffix automaton of P^r (the reversed pattern) is built. The algorithm searches backwards inside the text window for a substring of the pattern P using the suffix automaton. Each time a terminal state is reached before hitting the beginning of the window, the position inside the window is remembered. This corresponds to finding a *prefix* of the pattern equal to a suffix of the window (since the reverse suffixes of P^r are the prefixes of P). The last prefix recognized backwards is the *longest* prefix of P in the window. A match is found if the complete window is read, while the check is abandoned when there is no transition to follow in the automaton. In either case, the window is shifted to align with the longest prefix recognized. See Figure 8.19.

This algorithm is $O(mn)$ time in the worst case and $O(n \log(m)/m)$ on average. There exists also a multipattern version of this algorithm called MultiBDM, which is the fastest for many patterns or very long patterns.

BDM rarely beats the best BM algorithms. However, a recent bit-parallel implementation called BNDM improves over BM in a wide range of cases. This algorithm simulates the non-deterministic suffix automaton using bit-parallelism. The algorithm supports some extended patterns and other applications mentioned in Shift-Or, while keeping more efficient than Shift-Or.

8.5.6 Practical Comparison

Figure 8.20 shows a practical comparison between string matching algorithms run on our reference machine. The values are correct within 5% of accuracy with a 95% confidence interval. We tested English text from the TREC collection, DNA (corresponding to 'h.influenzae') and random text uniformly generated over 64 letters. The patterns were randomly selected from the text except for random

text, where they were randomly generated. We tested over 10 Mb of text and measured CPU time. We tested short patterns on English and random text and long patterns on DNA, which are the typical cases.

We first analyze the case of random text, where except for very short patterns the clear winners are BNDM (the bit-parallel implementation of BDM) and the BMS (Sunday) algorithm. The more classical Boyer-Moore and BDM algorithms are also very close. Among the algorithms that do not improve with the pattern length, Shift-Or is the fastest, and KMP is much slower than the naive algorithm.

The picture is similar for English text, except that we have included the Agrep software in this comparison, which worked well only on English text. Agrep turns out to be much faster than others. This is not because of using a special algorithm (it uses a BM-family algorithm) but because the code is carefully optimized. This shows the importance of careful coding as well as using good algorithms, especially in text searching where a few operations per text character are performed.

Longer patterns are shown for a DNA text. BNDM is the fastest for moderate patterns, but since it does not improve with the length after $m > w$, the classical BDM finally obtains better times. They are much better than the Boyer-Moore family because the alphabet is small and the suffix automaton technique makes better use of the information on the pattern.

We have not shown the case of extended patterns, that is, where flexibility plays a role. For this case, BNDM is normally the fastest when it can be applied (e.g., it supports classes of characters but not wild cards), otherwise Shift-Or is the best option. Shift-Or is also the best option when the text must be accessed sequentially and it is not possible to skip characters.

8.5.7 Phrases and Proximity

If a sequence of words is searched to appear in the text exactly as in the pattern (i.e., with the same separators) the problem is similar to that of exact search of a single pattern, by just forgetting the fact that there are many words. If any separator between words is to be allowed, it is possible to arrange it using an extended pattern or regular expression search.

The best way to search a phrase element-wise is to search for the element which is less frequent or can be searched faster (both criteria normally match). For instance, longer patterns are better than shorter ones; allowing fewer errors is better than allowing more errors. Once such an element is found, the neighboring words are checked to see if a complete match is found.

A similar algorithm can be used to search a proximity query.

8.6 Pattern Matching

We present in this section the main techniques to deal with complex patterns. We divide it into two main groups: searching allowing errors and searching for extended patterns.

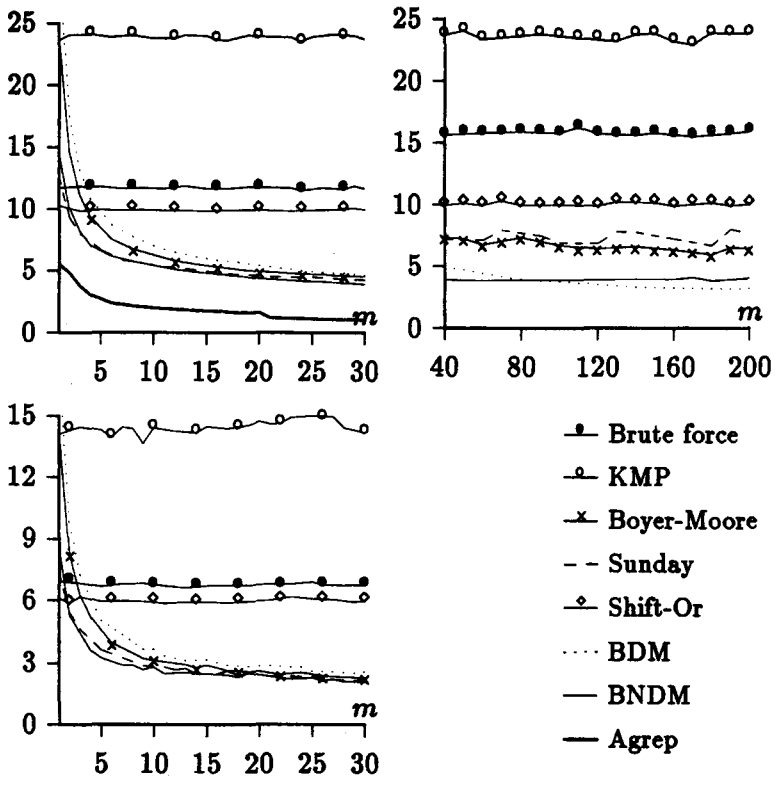


Figure 8.20 Practical comparison among algorithms. The upper left plot is for short patterns on English text. The upper right one is for long patterns on DNA. The lower plot is for short patterns on random text (on 64 letters). Times are in tenths of seconds per megabyte.

8.6.1 String Matching Allowing Errors

This problem (called 'approximate string matching') can be stated as follows: given a short pattern P of length m, a long text T of length n, and a maximum allowed number of errors k, find all the text positions where the pattern occurs with at most k errors. This statement corresponds to the Levenshtein distance. With minimal modifications it is adapted to searching whole words matching the pattern with k errors.

This problem is newer than exact string matching, although there are already a number of solutions. We sketch the main approaches.

Dynamic Programming

The classical solution to approximate string matching is based on dynamic programming. A matrix $C[0..m, 0..n]$ is filled column by column, where $C[i, j]$

represents the minimum number of errors needed to match $P_{1..i}$ to a suffix of $T_{1..j}$. This is computed as follows

$$\begin{aligned} C[0,j] &= 0 \\ C[i,0] &= i \\ C[i,j] &= \text{if } (P_i = T_j) \text{ then } C[i-1,j-1] \\ &\quad \text{else } 1 + \min(C[i-1,j], C[i,j-1], C[i-1,j-1]) \end{aligned}$$

where a match is reported at text positions j such that $C[m,j] \leq k$ (the final positions of the occurrences are reported).

Therefore, the algorithm is $O(mn)$ time. Since only the previous column of the matrix is needed, it can be implemented in $O(m)$ space. Its preprocessing time is $O(m)$. Figure 8.21 illustrates this algorithm.

In recent years several algorithms have been presented that achieve $O(kn)$ time in the worst case or even less in the average case, by taking advantage of the properties of the dynamic programming matrix (e.g., values in neighbor cells differ at most by one).

Automaton

It is interesting to note that the problem can be reduced to a non-deterministic finite automaton (NFA). Consider the NFA for $k = 2$ errors shown in Figure 8.22. Each row denotes the number of errors seen. The first one 0, the second one 1, and so on. Every column represents matching the pattern up to a given position. At each iteration, a new text character is read and the automaton changes its states. Horizontal arrows represent matching a character, vertical arrows represent insertions into the pattern, solid diagonal arrows represent replacements, and dashed diagonal arrows represent deletions in the pattern (they are ε-transitions). The automaton accepts a text position as the end of a match

		s	u	r	g	e	r	y
	0	0	0	0	0	0	0	0
s	1	0	1	1	1	1	1	1
u	2	1	0	1	2	2	2	2
r	3	2	1	0	1	2	2	3
v	4	3	2	1	1	2	3	3
e	5	4	3	2	2	1	2	3
y	6	5	4	3	3	**2**	**2**	**2**

Figure 8.21 The dynamic programming algorithm search '**survey**' in the text '**surgery**' with two errors. Bold entries indicate matching positions.

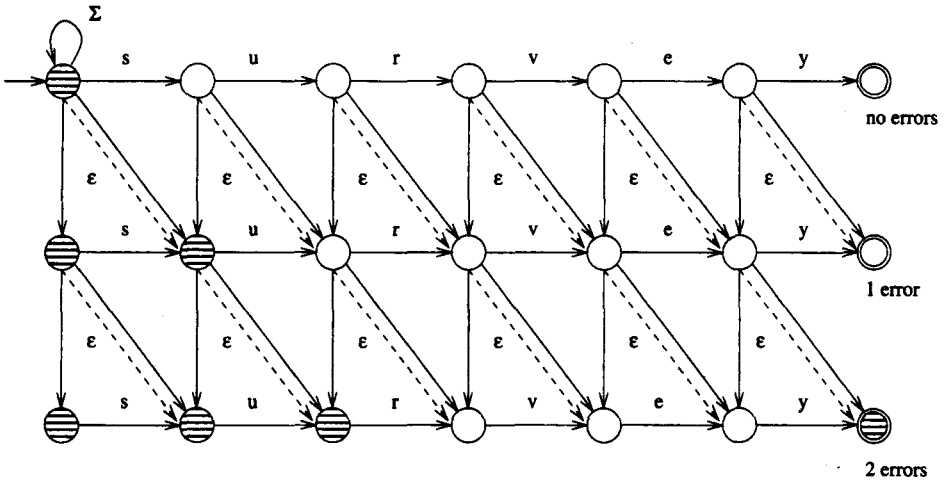


Figure 8.22 An NFA for approximate string matching of the pattern '**survey**' with two errors. The shaded states are those active after reading the text '**surgery**'. Unlabelled transitions match any character.

with k errors whenever the $(k+1)$-th rightmost state is active.

It is not hard to see that once a state in the automaton is active, all the states of the same column and higher rows are active too. Moreover, at a given text character, if we collect the smallest active rows at each column, we obtain the current column of the dynamic programming algorithm. Figure 8.22 illustrates this (compare the figure with Figure 8.21).

One solution is to make this automaton deterministic (DFA). Although the search phase is $O(n)$, the DFA can be huge. An alternative solution is based on bit-parallelism and is explained next.

Bit-Parallelism

Bit-parallelism has been used to parallelize the computation of the dynamic programming matrix (achieving average complexity $O(kn/w)$) and to parallelize the computation of the NFA (without converting it to deterministic), obtaining $O(kmn/w)$ time in the worst case. Such algorithms achieve $O(n)$ search time for short patterns and are currently the fastest ones in many cases, running at 6 to 10 Mb per second on our reference machine.

Filtering

Finally, other approaches first filter the text, reducing the area where dynamic programming needs to be used. These algorithms achieve 'sublinear' expected time in many cases for low error ratios (i.e., not all text characters are inspected,

$O(kn \log_\sigma(m)/m)$ is a typical figure), although the filtration is not effective for more errors. Filtration is based on the fact that some portions of the pattern must appear with no errors even in an approximate occurrence.

The fastest algorithm for low error levels is based on filtering: if the pattern is split into $k+1$ pieces, any approximate occurrence must contain at least one of the pieces with no errors, since k errors cannot alter all the $k+1$ pieces. Hence, the search begins with a multipattern exact search for the pieces and it later verifies the areas that may contain a match (using another algorithm).

8.6.2 Regular Expressions and Extended Patterns

General regular expressions are searched by building an automaton which finds all their occurrences in a text. This process first builds a non-deterministic finite automaton of size $O(m)$, where m is the length of the regular expression. The classical solution is to convert this automaton to deterministic form. A deterministic automaton can search any regular expression in $O(n)$ time. However, its size and construction time can be exponential in m, i.e. $O(m2^m)$. See Figure 8.23. Excluding preprocessing, this algorithm runs at 6 Mb/sec in the reference machine.

Recently the use of bit-parallelism has been proposed to avoid the construction of the deterministic automaton. The non-deterministic automaton is simulated instead. One bit per automaton state is used to represent whether the state is active or not. Due to the algorithm used to build the non-deterministic automaton, all the transitions move forward except for ε-transitions. The idea is that for each text character two steps are carried out. The first one moves forward, and the second one takes care of all the ε-transitions. A function E from bit masks to bit masks is precomputed so that all the corresponding bits are moved according to the ε-transitions. Since this function is very large (i.e., 2^m entries) its domain is split in many functions from 8- or 16-bit submasks to m-bit masks. This is possible because $E(B_1...B_j) = E(B_1)|...|E(B_j)$, where B_i

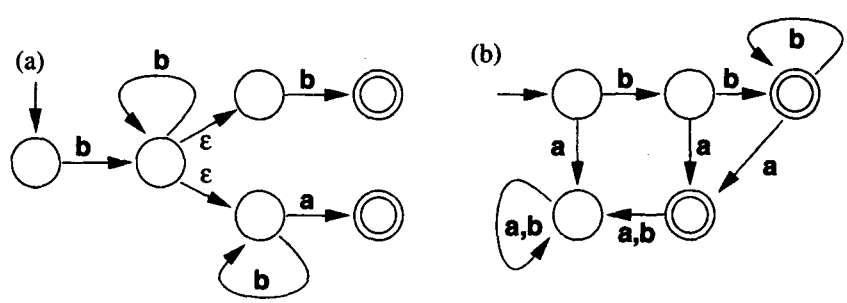


Figure 8.23 The non-deterministic (a) and deterministic (b) automata for the regular expression `b b*` (`b` | `b*a`).

are the submasks. Hence, the scheme performs $\lceil m/8 \rceil$ or $\lceil m/16 \rceil$ operations per text character and needs $\lceil m/8 \rceil 2^8 \lceil m/w \rceil$ or $\lceil m/16 \rceil 2^{16} \lceil m/w \rceil$ machine words of memory.

Extended patterns can be rephrased as regular expressions and solved as before. However, in many cases it is more efficient to give them a specialized solution, as we saw for the extensions of exact searching (bit-parallel algorithms). Moreover, extended patterns can be combined with approximate search for maximum flexibility. In general, the bit-parallel approach is the best equipped to deal with extended patterns.

Real times for regular expressions and extended pattern searching using this technique are between 2–8 Mb/sec.

8.6.3 Pattern Matching Using Indices

We end this section by explaining how the indexing techniques we presented for simple searching of words can in fact be extended to search for more complex patterns.

Inverted Files

As inverted files are word-oriented, other types of queries such as suffix or substring queries, searching allowing errors and regular expressions, are solved by a *sequential* (i.e., online) search over the vocabulary. This is not too bad since the size of the vocabulary is small with respect to the text size.

After either type of search, a list of vocabulary words that matched the query is obtained. All their lists of occurrences are now *merged* to retrieve a list of documents and (if required) the matching text positions. If block addressing is used and the positions are required or the blocks do not coincide with the retrieval unit, the search must be completed with a sequential search over the blocks.

Notice that an inverted index is word-oriented. Because of that it is not surprising that it is not able to efficiently find approximate matches or regular expressions that span many words. This is a restriction of this scheme. Variations that are not subject to this restriction have been proposed for languages which do not have a clear concept of word, like Finnish. They collect text *samples* or *n-grams*, which are fixed-length strings picked at regular text intervals. Searching is in general more powerful but more expensive.

In a full-inverted index, search times for simple words allowing errors on 250 Mb of text took our reference machine from 0.6 to 0.85 seconds, while very complex expressions on extended patterns took from 0.8 to 3 seconds. As a comparison, the same collection cut in blocks of 1 Mb size takes more than 8 seconds for an approximate search with one error and more than 20 for two errors.

Suffix Trees and Suffix Arrays

If the suffix tree indexes all text positions it can search for words, prefixes, suffixes and substrings with the same search algorithm and cost described for word search. However, indexing all positions makes the index 10 to 20 times the text size for suffix trees.

Range queries are easily solved too, by just searching both extremes in the trie and then collecting all the leaves which lie in the middle. In this case the cost is the height of the tree, which is $O(\log n)$ on average (excluding the tasks of collecting and sorting the leaves).

Regular expressions can be searched in the suffix tree. The algorithm simply simulates sequential searching of the regular expression. It begins at the root, since any possible match starts there too. For each child of the current node labeled by the character c, it assumes that the next text character is c and recursively enters into that subtree. This is done for each of the children of the current node. The search stops only when the automaton has no transition to follow. It has been shown that for random text only $O(n^{\alpha}\text{polylog}(n))$ nodes are traversed (for $0 < \alpha < 1$ dependent on the regular expression). Hence, the search time is sublinear for regular expressions *without* the restriction that they must occur inside a word. Extended patterns can be searched in the same way by taking them as regular expressions.

Unrestricted approximate string matching is also possible using the same idea. We present a simplified version here. Imagine that the search is online and traverse the tree recursively as before. Since all suffixes start at the root, any match starts at the root too, and therefore do not allow the match to start later. The search will automatically stop at depth $m + k$ at most (since at that point more than k errors have occurred). This implies constant search time if n is large enough (albeit exponential on m and k). Other problems such as approximate search of extended patterns can be solved in the same way, using the appropriate online algorithm.

Suffix trees are able to perform other complex searches that we have not considered in our query language (see Chapter 4). These are specialized operations which are useful in specific areas. Some examples are: find the longest substring in the text that appears more than once, find the most common substring of a fixed size, etc.

If a suffix array indexes all text positions, *any* algorithm that works on suffix trees at $C(n)$ cost will work on suffix arrays at $O(C(n) \log n)$ cost. This is because the operations performed on the suffix tree consist of descending to a child node, which is done in $O(1)$ time. This operation can be simulated in the suffix array in $O(\log n)$ time by binary searching the new boundaries (each suffix tree node corresponds to a string, which can be mapped to the suffix array interval holding all suffixes starting with that string). Some patterns can be searched directly in the suffix array in $O(\log n)$ total search time without simulating the suffix tree. These are: word, prefix, suffix and subword search, as well as range search.

However, again, indexing all text positions normally makes the suffix array

size four times or more the text size. A different alternative for suffix arrays is to index only word beginnings and to use a vocabulary supra-index, using the same search algorithms used for the inverted lists.

8.7 Structural Queries

The algorithms to search on structured text (see Chapter 4) are largely dependent on each model. We extract their common features in this section.

A first concern about this problem is how to store the structural information. Some implementations build an ad hoc index to store the structure. This is potentially more efficient and independent of any consideration about the text. However, it requires extra development and maintenance effort.

Other techniques assume that the structure is marked in the text using 'tags' (i.e., strings that identify the structural elements). This is the case with HTML text but not the case with C code where the marks are implicit and are inherent to C. The technique relies on the same index to query content (such as inverted files), using it to index and search those tags as if they were words. In many cases this is as efficient as an ad hoc index, and its integration into an existing text database is simpler. Moreover, it is possible to define the structure dynamically, since the appropriate tags can be selected at search time. For that goal, inverted files are better since they naturally deliver the results in text order, which makes the structure information easier to obtain. On the other hand, some queries such as direct ancestry are hard to answer without an ad hoc index.

Once the content and structural elements have been found by using some index, a set of answers is generated. The models allow further operations to be applied on those answers, such as 'select all areas in the left-hand argument which contain an area of the right-hand argument.' This is in general solved in a way very similar to the set manipulation techniques already explained in section 8.4. However, the operations tend to be more complex, and it is not always possible to find an evaluation algorithm which has linear time with respect to the size of the intermediate results.

It is worth mentioning that some models use completely different algorithms, such as exhaustive search techniques for tree pattern matching. Those problems are NP-complete in many cases.

8.8 Compression

In this section we discuss the issues of searching compressed text directly and of searching compressed indices. Compression is important when available storage is a limiting factor, as is the case of indexing the Web.

Searching and compression were traditionally regarded as exclusive operations. Texts which were not to be searched could be compressed, and to search

a compressed text it had to be decompressed first. In recent years, very efficient compression techniques have appeared that allow searching directly in the compressed text. Moreover, the search performance is *improved*, since the CPU times are similar but the disk times are largely reduced. This leads to a win-win situation.

Discussion on how common text and lists of numbers can be compressed has been covered in Chapter 7.

8.8.1 Sequential Searching

A few approaches to directly searching compressed text exist. One of the most successful techniques in practice relies on Huffman coding taking words as symbols. That is, consider each different text word as a symbol, count their frequencies, and generate a Huffman codefor the words. Then, compress the text by replacing each word with its code. To improve compression/decompression efficiency, the Huffman code uses an alphabet of bytes instead of bits. This scheme compresses faster and better than known commercial systems, even those based on Ziv-Lempel coding.

Since Huffman coding needs to store the codes of each symbol, this scheme has to store the whole vocabulary of the text, i.e. the list of all different text words. This is fully exploited to efficiently search complex queries. Although according to Heaps' law the vocabulary (i.e., the alphabet) grows as $O(n^{\beta})$ for $0 < \beta < 1$, the generalized Zipf's law shows that the distribution is skewed enough so that the entropy remains constant (i.e., the compression ratio will not degrade as the text grows). Those laws are explained in Chapter 6.

Any single-word or pattern query is first searched in the vocabulary. Some queries can be binary searched, while others such as approximate searching or regular expression searching must traverse sequentially all the vocabulary. This vocabulary is rather small compared to the text size, thanks to Heaps' law. Notice that this process is exactly the same as the vocabulary searching performed by inverted indices, either for simple or complex pattern matching.

Once that search is complete, the list of different words that match the query is obtained. The Huffman codes of all those words are collected and they are searched in the compressed text. One alternative is to traverse byte-wise the compressed text and traverse the Huffman decoding tree in synchronization, so that each time that a leaf is reached, it is checked whether the leaf (i.e., word) was marked as 'matching' the query or not. This is illustrated in Figure 8.24. Boyer-Moore filtering can be used to speed up the search.

Solving phrases is a little more difficult. Each element is searched in the vocabulary. For each word of the vocabulary we define a bit mask. We set the i-th bit in the mask of all words which match with the i-th element of the phrase query. This is used together with the Shift-Or algorithm. The text is traversed byte-wise, and only when a leaf is reached, does the Shift-Or algorithm consider that a new text symbol has been read, whose bit mask is that of the leaf (see Figure 8.24). This algorithm is surprisingly simple and efficient.

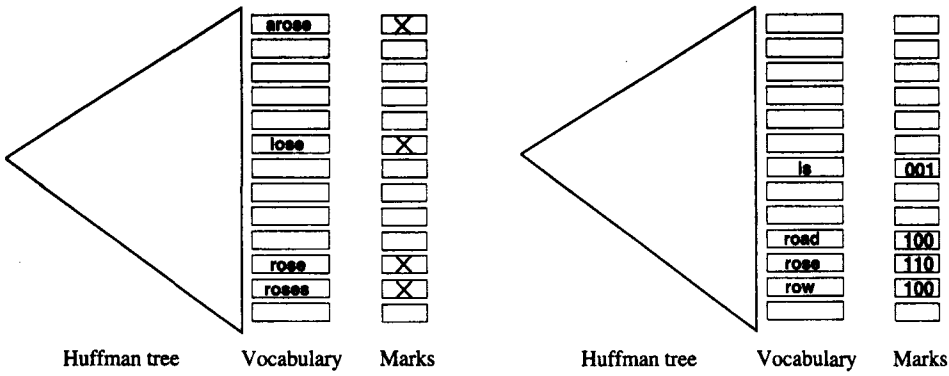


Figure 8.24 On the left, searching for the simple pattern '`rose`' allowing one error. On the right, searching for the phrase '`ro* rose is`,' where '`ro*`' represents a prefix search.

This scheme is especially fast when it comes to solving a complex query (regular expression, extended pattern, approximate search, etc.) that would be slow with a normal algorithm. This is because the complex search is done only in the small vocabulary, after which the algorithm is largely insensitive to the complexity of the originating query. Its CPU times for a simple pattern are slightly higher than those of Agrep (briefly described in section 8.5.6). However, if the I/O times are considered, compressed searching is faster than all the online algorithms. For complex queries, this scheme is unbeaten by far.

On the reference machine, the CPU times are 14 Mb/sec for any query, while for simple queries this improves to 18 Mb/sec if the speedup technique is used. Agrep, on the other hand, runs at 15 Mb/sec on simple searches and at 1–4 Mb/sec for complex ones. Moreover, I/O times are reduced to one third on the compressed text.

8.8.2 Compressed Indices

Inverted Files

Inverted files are quite amenable to compression. This is because the lists of occurrences are in increasing order of text position. Therefore, an obvious choice is to represent the differences between the previous position and the current one. These differences can be represented using less space by using techniques that favor small numbers (see Chapter 7). Notice that, the longer the lists, the smaller the differences. Reductions in 90% for block-addressing indices with blocks of 1 Kb size have been reported.

It is important to notice that compression does not necessarily degrade time performance. Most of the time spent in answering a query is in the disk transfer. Keeping the index compressed allows the transference of less data, and it may be worth the CPU work of decompressing. Notice also that the lists of

occurrences are normally traversed in a sequential manner, which is not affected by a differential compression. Query times on compressed or decompressed indices are reported to be roughly similar.

The text can also be compressed independently of the index. The text will be decompressed only to display it, or to traverse it in case of block addressing. Notice in particular that the online search technique described for compressed text in section 8.8.1 uses a vocabulary. It is possible to integrate both techniques (compression and indexing) such that they share the same vocabulary for both tasks and they do not decompress the text to index or to search.

Suffix Trees and Suffix Arrays

Some efforts to compress suffix trees have been pursued. Important reductions of the space requirements have been obtained at the cost of more expensive searching. However, the reduced space requirements happen to be similar to those of uncompressed suffix arrays, which impose much smaller performance penalties.

Suffix arrays are very hard to compress further. This is because they represent an almost perfectly random permutation of the pointers to the text.

However, the subject of building suffix arrays on compressed text has been pursued. Apart from reduced space requirements (the index plus the compressed text take less space than the uncompressed text), the main advantage is that both index construction and querying almost double their performance. Construction is faster because more compressed text fits in the same memory space, and therefore fewer text blocks are needed. Searching is faster because a large part of the search time is spent in disk seek operations over the text area to compare suffixes. If the text is smaller, the seeks reduce proportionally.

A compression technique very similar to that shown in section 8.8.1 is used. However, the Huffman code on words is replaced by a Hu-Tucker coding. The Hu-Tucker code respects the lexicographical relationships between the words, and therefore direct binary search over the compressed text is possible (this is necessary at construction and search time). This code is suboptimal by a very small percentage (2–3% in practice, with an analytical upper bound of 5%).

Indexing times for 250 Mb of text on the reference machine are close to 1.6 Mb/min if compression is used, while query times are reduced to 0.5 seconds in total and 0.3 seconds for the text alone. Supra-indices should reduce the total search time to 0.15 seconds.

Signature Files

There are many alternative ways to compress signature files. All of them are based on the fact that only a few bits are set in the whole file. It is then possible

to use efficient methods to code the bits which are not set, for instance run-length encoding. Different considerations arise if the file is stored as a sequence of bit masks or with one file per bit of the mask. They allow us to reduce space and hence disk times, or alternatively to increase B (so as to reduce the false drop probability) keeping the same space overhead. Compression ratios near 70% are reported.

8.9 Trends and Research Issues

In this chapter we covered extensively the current techniques of dealing with text retrieval. We first covered indices and then online searching. We then reviewed set manipulation, complex pattern matching and finally considered compression techniques. Figure 8.25 summarizes the tradeoff between the space needed for the index and the time to search one single word.

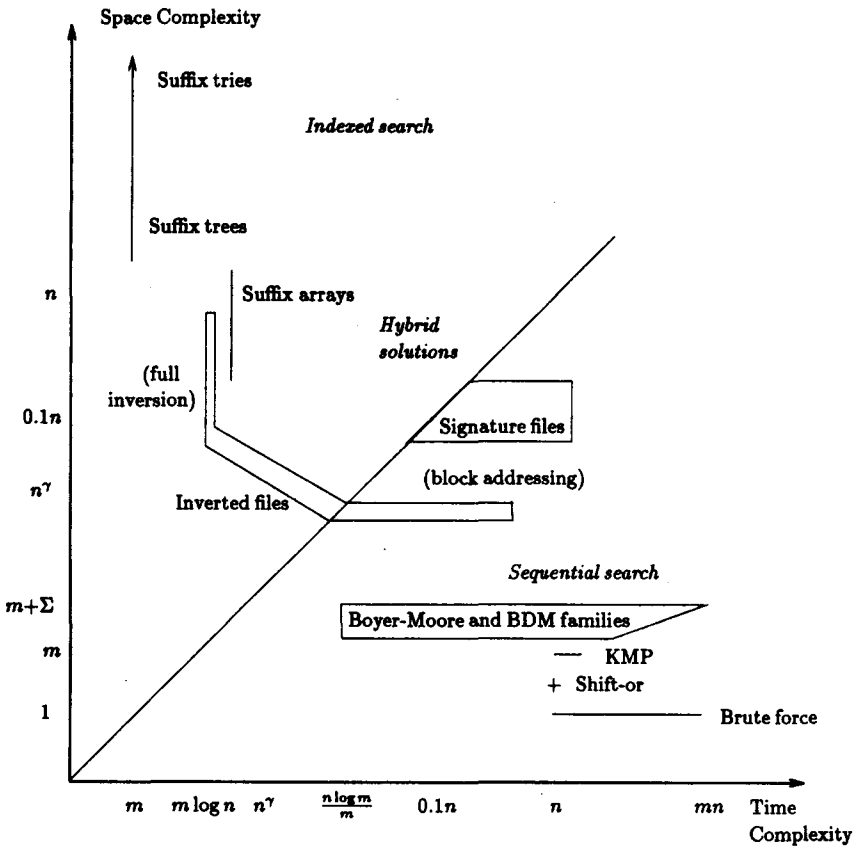


Figure 8.25 Tradeoff of index space versus word searching time.

Probably the most adequate indexing technique in practice is the inverted file. As we have shown throughout the chapter, many hidden details in other structures make them harder to use and less efficient in practice, as well as less flexible for dealing with new types of queries. These structures, however, still find application in restricted areas such as genetic databases (for suffix trees and arrays, for the relatively small texts used and their need to pose specialized queries) or some office systems (for signature files, because the text is rarely queried in fact).

The main trends in indexing and searching textual databases today are

- **Text collections are becoming huge**. This poses more demanding requirements at all levels, and solutions previously affordable are not any more. On the other hand, the speed of the processors and the relative slowness of external devices have changed what a few years ago were reasonable options (e.g., it is better to keep a text compressed because reading less text from disk and decompressing in main memory pays off).
- **Searching is becoming more complex**. As the text databases grow and become more heterogeneous and error-prone, enhanced query facilities are required, such as exploiting the text structure or allowing errors in the text. Good support for extended queries is becoming important in the evaluation of a text retrieval system.
- **Compression is becoming a star in the field**. Because of the changes mentioned in the time cost of processors and external devices, and because of new developments in the area, text retrieval and compression are no longer regarded as disjoint activities. Direct indexing and searching on compressed text provides better (sometimes much better) time performance and less space overhead at the same time. Other techniques such as block addressing trade space for processor time.

8.10 Bibliographic Discussion

A detailed explanation of a full inverted index and its construction and querying process can be found in [26]. This work also includes an analysis of the algorithms on inverted lists using the distribution of natural language. The in-place construction is described in [572]. Another construction algorithm is presented in [341].

The idea of block addressing inverted indices was first presented in a system called Glimpse [540], which also first exposed the idea of performing complex pattern matching using the vocabulary of the inverted index. Block addressing indices are analyzed in [42], where some performance improvements are proposed. The variant that indexes sequences instead of words has been implemented in a system called Grampse, which is described in [497].

Suffix arrays were presented in [538] together with the algorithm to build them in $O(n \log n)$ character comparisons. They were independently discovered