# Chapter 4
# Query Languages

---

## with Gonzalo Navarro

## 4.1 Introduction

We cover in this chapter the different kinds of queries normally posed to text retrieval systems. This is in part dependent on the retrieval model the system adopts, i.e., a full-text system will not answer the same kinds of queries as those answered by a system based on keyword ranking (as Web search engines) or on a hypertext model. In Chapter 8 we explain *how* the user queries are solved, while in this chapter we show *which* queries can be formulated. The type of query the user might formulate is largely dependent on the underlying information retrieval model. The different models for text retrieval systems are covered in Chapter 2.

As in previous chapters, we want to distinguish between information retrieval and data retrieval, as we use this dichotomy to classify different query languages. We have chosen to distinguish first languages that allow the answer to be ranked, that is, languages for information retrieval. As covered in Chapter 2, for the basic information retrieval models, keyword-based retrieval is the main type of querying task. For query languages not aimed at information retrieval, the concept of ranking cannot be easily defined, so we consider them as languages for data retrieval. Furthermore, some query languages are not intended for final users and can be viewed as languages that a higher level software package should use to query an on-line database or a CD-ROM archive. In that case, we talk about *protocols* rather than query languages. Depending on the user experience, a different query language will be used. For example, if the user knows exactly what he wants, the retrieval task is easier and ranking may not even be needed.

An important issue is that most query languages try to use the content (i.e., the semantics) and the structure of the text (i.e., the text syntax) to find relevant documents. In that sense, the system may fail to find the relevant answers (see Chapter 3). For this reason, a number of techniques meant to enhance the usefulness of the queries exist. Examples include the expansion of a word to the set of its synonyms or the use of a thesaurus and stemming to

put together all the derivatives of the same word. Moreover, some words which are very frequent and do not carry meaning (such as 'the'), called *stopwords*, may be removed. This subject is covered in Chapter 7. Here we assume that all the query preprocessing has already been done. Although these operations are usually done for information retrieval, many of them can also be useful in a data retrieval context. When we want to emphasize the difference between words that can be retrieved by a query and those which cannot, we call the former 'keywords.'

Orthogonal to the kind of queries that can be asked is the subject of the *retrieval unit* the information system adopts. The retrieval unit is the basic element which can be retrieved as an answer to a query (normally a set of such basic elements is retrieved, sometimes ranked by relevance or other criterion). The retrieval unit can be a file, a document, a Web page, a paragraph, or some other structural unit which contains an answer to the search query. From this point on, we will simply call those retrieval units 'documents,' although as explained this can have different meanings (see also Chapter 2).

This chapter is organized as follows. We first show the queries that can be formulated with keyword-based query languages. They are aimed at information retrieval, including simple words and phrases as well as Boolean operators which manipulate sets of documents. In the second section we cover pattern matching, which includes more complex queries and is generally aimed at complementing keyword searching with more powerful data retrieval capabilities. Third, we cover querying on the structure of the text, which is more dependent on the particular text model. Finally, we finish with some standard protocols used on the Internet and by CD-ROM publishers.

## 4.2 Keyword-Based Querying

A query is the formulation of a user information need. In its simplest form, a query is composed of keywords and the documents containing such keywords are searched for. Keyword-based queries are popular because they are intuitive, easy to express, and allow for fast ranking. Thus, a query can be (and in many cases is) simply a word, although it can in general be a more complex combination of operations involving several words.

In the rest of this chapter we will refer to single-word and multiple-word queries as *basic queries*. Patterns, which are covered in section 4.3, are also considered as basic queries.

### 4.2.1 Single-Word Queries

The most elementary query that can be formulated in a text retrieval system is a word. Text documents are assumed to be essentially long sequences of words. Although some models present a more general view, virtually all models allow us

to see the text in this perspective and to search words. Some models are also able to see the internal division of words into letters. These latter models permit the searching of other types of patterns, which are covered in section 4.3. The set of words retrieved by these extended queries can then be fed into the word-treating machinery, say to perform thesaurus expansion or for ranking purposes.

A word is normally defined in a rather simple way. The alphabet is split into 'letters' and 'separators,' and a word is a sequence of letters surrounded by separators. More complex models allow us to specify that some characters are not letters but do not split a word, e.g. the hyphen in 'on-line.' It is good practice to leave the choice of what is a letter and what is a separator to the manager of the text database.

The division of the text into words is not arbitrary, since words carry a lot of meaning in natural language. Because of that, many models (such as the vector model) are completely structured on the concept of words, and words are the only type of queries allowed (moreover, some systems only allow a small set of words to be extracted from the documents). The result of word queries is the set of documents containing at least one of the words of the query. Further, the resulting documents are ranked according to a degree of similarity to the query. To support ranking, two common statistics on word occurrences inside texts are commonly used: 'term frequency' which counts the number of times a word appears inside a document and 'inverse document frequency' which counts the number of documents in which a word appears. See Chapter 2 for more details.

Additionally, the exact positions where a word appears in the text may be required for instance, by an interface which highlights each occurrence of that word.

### 4.2.2 Context Queries

Many systems complement single-word queries with the ability to search words in a given *context*, that is, near other words. Words which appear near each other may signal a higher likelihood of relevance than if they appear apart. For instance, we may want to form phrases of words or find words which are proximal in the text. Therefore, we distinguish two types of queries:

- **Phrase** is a sequence of single-word queries. An occurrence of the phrase is a sequence of words. For instance, it is possible to search for the word 'enhance,' and then for the word 'retrieval.' In phrase queries it is normally understood that the separators in the text need not be the same as those in the query (e.g., two spaces versus one space), and uninteresting words are not considered at all. For instance, the previous example could match a text such as '...enhance the retrieval...'. Although the notion of a phrase is a very useful feature in most cases, not all systems implement it.
- **Proximity** A more relaxed version of the phrase query is the proximity query. In this case, a sequence of single words or phrases is given, together

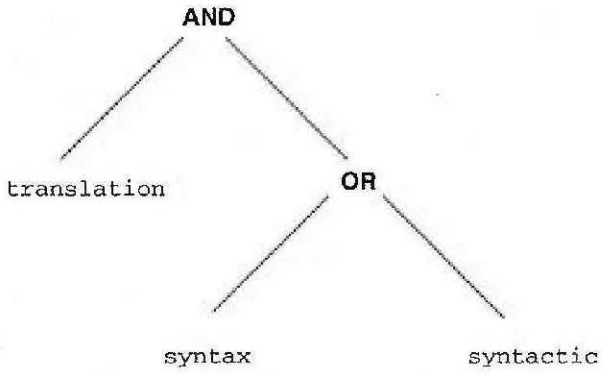



**Figure 4.1** An example of a query syntax tree. It will retrieve all the documents which contain the word '`translation`' as well as either the word '`syntax`' or the word '`syntactic`'.

with a maximum allowed distance between them. For instance, the above example could state that the two words should occur within four words, and therefore a match could be '`...enhance the power of retrieval...`' This distance can be measured in characters or words depending on the system. The words and phrases may or may not be required to appear in the same order as in the query.

Phrases can be ranked in a fashion somewhat analogous to single words (see Chapters 2 and 5 for details). Proximity queries can be ranked in the same way if the parameters used by the ranking technique do not depend on physical proximity. Although it is not clear how to do better ranking, physical proximity has semantic value. This is because in most cases the proximity means that the words are in the same paragraph, and hence related in some way.

### 4.2.3 Boolean Queries

The oldest (and still heavily used) form of combining keyword queries is to use Boolean operators. A *Boolean query* has a syntax composed of *atoms* (i.e., basic queries) that retrieve documents, and of *Boolean operators* which work on their operands (which are sets of documents) and deliver sets of documents. Since this scheme is in general *compositional* (i.e., operators can be composed over the results of other operators), a *query syntax tree* is naturally defined, where the leaves correspond to the basic queries and the internal nodes to the operators. The query syntax tree operates on an algebra over sets of documents (and the final answer of the query is also a set of documents). This is much as, for instance, the syntax trees of arithmetic expressions where the numbers and variables are the leaves and the operations form the internal nodes. Figure 4.1 shows an example.

The operators most commonly used, given two basic queries or Boolean

subexpressions $e_1$ and $e_2$, are:

- **OR** The query ($e_1$ OR $e_2$) selects all documents which satisfy $e_1$ or $e_2$. Duplicates are eliminated.
- **AND** The query ($e_1$ AND $e_2$) selects all documents which satisfy both $e_1$ and $e_2$.
- **BUT** The query ($e_1$ BUT $e_2$) selects all documents which satisfy $e_1$ but not $e_2$. Notice that classical Boolean logic uses a NOT operation, where (NOT $e_2$) is valid whenever $e_2$ is not. In this case all documents not satisfying $e_2$ should be delivered, which may retrieve a huge amount of text and is probably not what the user wants. The BUT operator, instead, restricts the universe of retrievable elements to the result of $e_1$.†

Besides selecting the appropriate documents, the IR system may also sort the documents by some criterion, highlight the occurrences within the documents of the words mentioned in the query, and allow feedback by taking the answer set as a basis to reformulate the query.

With classic Boolean systems, no ranking of the retrieved documents is normally provided. A document either satisfies the Boolean query (in which case it is retrieved) or it does not (in which case it is not retrieved). This is quite a limitation because it does not allow for partial matching between a document and a user query. To overcome this limitation, the condition for retrieval must be relaxed. For instance, a document which partially satisfies an AND condition might be retrieved.

In fact, it is widely accepted that users not trained in mathematics find the meaning of Boolean operators difficult to grasp. With this problem in mind, a 'fuzzy Boolean' set of operators has been proposed. The idea is that the meaning of AND and OR can be relaxed, such that instead of forcing an element to appear in *all* the operands (AND) or at least in *one* of the operands (OR), they retrieve elements appearing in *some* operands (the AND may require it to appear in more operands than the OR). Moreover, the documents are ranked higher when they have a larger number of elements in common with the query (see Chapter 2).

### 4.2.4 Natural Language

Pushing the fuzzy Boolean model even further, the distinction between AND and OR can be completely blurred, so that a query becomes simply an enumeration of words and context queries. All the documents matching a portion of the user query are retrieved. Higher ranking is assigned to those documents matching more parts of the query. The negation can be handled by letting the user express

† Notice that the same problem arises in the relational calculus, which is shown similar to the relational algebra only when 'unsafe' expressions are avoided. Unsafe expressions are those that make direct or indirect reference to a universe of elements, as NOT does.

that some words are not desired, so that the documents containing them are penalized in the ranking computation. A threshold may be selected so that the documents with very low weights are not retrieved. Under this scheme we have completely eliminated any reference to Boolean operations and entered into the field of natural language queries. In fact, one can consider that Boolean queries are a simplified abstraction of natural language queries.

A number of new issues arise once this model is used, especially those related to the proper way to rank an element with respect to a query. The search criterion can be re-expressed using a different model, where documents and queries are considered just as a vector of 'term weights' (with one coordinate per interesting keyword or even per existing text word) and queries are considered in exactly the same way (context queries are not considered in this case). Therefore, the query is now internally converted into a vector of term weights and the aim is to retrieve all the vectors (documents) which are *close* to the query (where closeness has to be defined in the model). This allows many interesting possibilities, for instance a complete document can be used as a query (since it is also a vector), which naturally leads to the use of relevance feedback techniques (i.e., the user can select a document from the result and submit it as a new query to retrieve documents similar to the selected one). The algorithms for this model are totally different from those based on searching patterns (it is even possible that not every text word needs to be searched but only a small set of hopefully representative keywords extracted from each document). Natural language querying is also covered in Chapter 14.

## 4.3 Pattern Matching

In this section we discuss more specific query formulations (based on the concept of a *pattern*) which allow the retrieval of pieces of text that have some property. These data retrieval queries are useful for linguistics, text statistics, and data extraction. Their result can be fed into the composition mechanism described above to form phrases and proximity queries, comprising what we have called *basic queries*. Basic queries can be combined using Boolean expressions. In this sense we can view these data retrieval capabilities as enhanced tools for information retrieval. However, it is more difficult to rank the result of a pattern matching expression.

A *pattern* is a set of syntactic features that must occur in a text segment. Those segments satisfying the pattern specifications are said to 'match' the pattern. We are interested in documents containing segments which match a given search pattern. Each system allows the specification of some types of patterns, which range from very simple (for example, words) to rather complex (such as regular expressions). In general, as more powerful is the set of patterns allowed, more involved are the queries that the user can formulate and more complex is the implementation of the search. The most used types of patterns are:

- **Words** A string (sequence of characters) which must be a word in the text (see section 4.2). This is the most basic pattern.
- **Prefixes** A string which must form the beginning of a text word. For instance, given the prefix '`comput`' all the documents containing words such as '`computer`,' '`computation`,' '`computing`,' etc. are retrieved.
- **Suffixes** A string which must form the termination of a text word. For instance, given the suffix '`ters`' all the documents containing words such as '`computers`,' '`testers`,' '`painters`,' etc. are retrieved.
- **Substrings** A string which can appear within a text word. For instance, given the substring '`tal`' all the documents containing words such as '`coastal`,' '`talk`,' '`metallic`,' etc. are retrieved. This query can be restricted to find the substrings inside words, or it can go further and search the substring anywhere in the text (in this case the query is not restricted to be a sequence of letters but can contain word separators). For instance, a search for '`any flow`' will match in the phrase '`...many flowers...`'
- **Ranges** A pair of strings which matches any word lying between them in lexicographical order. Alphabets are normally sorted, and this induces an order into the strings which is called *lexicographical order* (this is indeed the order in which words in a dictionary are listed). For instance, the range between words '`held`' and '`hold`' will retrieve strings such as '`hoax`' and '`hissing`.'
- **Allowing errors** A word together with an error threshold. This search pattern retrieves all text words which are 'similar' to the given word. The concept of similarity can be defined in many ways. The general concept is that the pattern or the text may have errors (coming from typing, spelling, or from optical character recognition software, among others), and the query should try to retrieve the given word and what are likely to be its erroneous variants. Although there are many models for similarity among words, the most generally accepted in text retrieval is the *Levenshtein distance*, or simply *edit distance*. The edit distance between two strings is the minimum number of character insertions, deletions, and replacements needed to make them equal (see Chapter 6). Therefore, the query specifies the maximum number of allowed errors for a word to match the pattern (i.e., the maximum allowed edit distance). This model can also be extended to search substrings (not only words), retrieving any text segment which is at the allowed edit distance from the search pattern. Under this extended model, if a typing error splits '`flower`' into '`flo wer`' it could still be found with one error, while in the restricted case of words it could not (since neither '`flo`' nor '`wer`' are at edit distance 1 from '`flower`'). Variations on this distance model are of use in computational biology for searching on DNA or protein sequences as well as in signal processing.
- **Regular expressions** Some text retrieval systems allow searching for *regular expressions*. A regular expression is a rather general pattern built

up by simple strings (which are meant to be matched as substrings) and the following operators:

- union: if $e_1$ and $e_2$ are regular expressions, then $(e_1|e_2)$ matches what $e_1$ or $e_2$ matches.
- concatenation: if $e_1$ and $e_2$ are regular expressions, the occurrences of $(e_1\ e_2)$ are formed by the occurrences of $e_1$ immediately followed by those of $e_2$ (therefore simple strings can be thought of as a concatenation of their individual letters).
- repetition: if $e$ is a regular expression, then $(e^*)$ matches a sequence of zero or more contiguous occurrences of $e$.

For instance, consider a query like 'pro (blem | tein) (s | $\varepsilon$) (0 | 1 | 2)*' (where $\varepsilon$ denotes the empty string). It will match words such as 'problem02' and 'proteins.' As in previous cases, the matches can be restricted to comprise a whole word, to occur inside a word, or to match an arbitrary text segment. This can also be combined with the previous type of patterns to search a regular expression allowing errors.

- **Extended patterns** It is normal to use a more user-friendly query language to represent some common cases of regular expressions. Extended patterns are subsets of the regular expressions which are expressed with a simpler syntax. The retrieval system can internally convert extended patterns into regular expressions, or search them with specific algorithms. Each system supports its own set of extended patterns, and therefore no formal definition exists. Some examples found in many new systems are:

  - classes of characters, i.e. one or more positions within the pattern are matched by any character from a pre-defined set. This involves features such as case-insensitive matching, use of ranges of characters (e.g., specifying that some character must be a digit), complements (e.g., some character must not be a letter), enumeration (e.g., a character must be a vowel), wild cards (i.e., a position within the pattern matches with anything), among others.
  - conditional expressions, i.e., a part of the pattern may or may not appear.
  - wild characters which match any sequence in the text, e.g. any word which starts as 'flo' and ends with 'ers,' which matches 'flowers' as well as 'flounders.'
  - combinations that allow some parts of the pattern to match exactly and other parts with errors.

## 4.4 Structural Queries

Up to now we have considered the text collection as a set of documents which can be queried with regard to their text content. This model is unable to take advantage of novel text features which are becoming commonplace, such as the

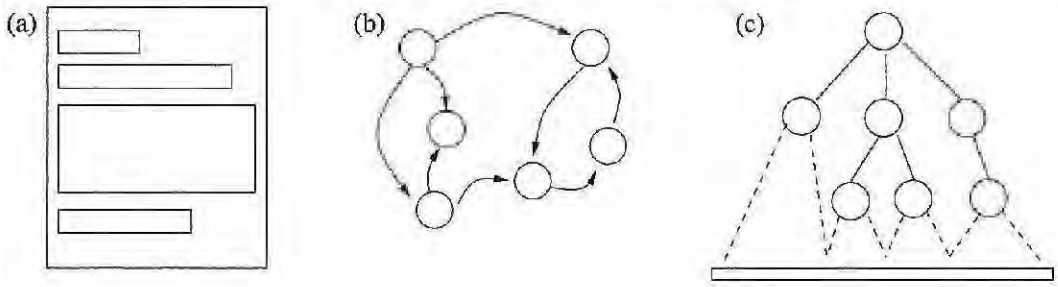



**Figure 4.2** The three main structures: (*a*) form-like fixed structure, (*b*) hypertext structure, and (*c*) hierarchical structure.

text structure. The text collections tend to have some structure built into them, and allowing the user to query those texts based on their structure (and not only their content) is becoming attractive. The standardization of languages to represent structured texts such as HTML has pushed forward in this direction (see Chapter 6).

As discussed in Chapter 2, mixing contents and structure in queries allows us to pose very powerful queries, which are much more expressive than each query mechanism by itself. By using a query language that integrates both types of queries, the retrieval quality of textual databases can be improved.

This mechanism is built on top of the basic queries, so that they select a set of documents that satisfy certain constraints on their content (expressed using words, phrases, or patterns that the documents must contain). On top of this, some structural constraints can be expressed using containment, proximity, or other restrictions on the structural elements (e.g., chapters, sections, etc.) present in the documents. The Boolean queries can be built on top of the structural queries, so that they combine the sets of documents delivered by those queries. In the Boolean syntax tree (recall the example of Figure 4.1) the structural queries form the leaves of the tree. On the other hand, structural queries can themselves have a complex syntax.

We divide this section according to the type of structures found in text databases. Figure 4.2 illustrates them. Although structured query languages should be amenable for ranking, this is still an open problem.

In what follows it is important to distinguish the difference between the structure that a text may *have* and what can be *queried* about that structure. In general, natural language texts may have any desired structure. However, different models allow the querying of only some aspects of the real structure. When we say that the structure allowed is restricted in some way, we mean that only the aspects which follow this restriction can be queried, albeit the text may have more structural information. For instance, it is possible that an article has a nested structure of sections and subsections, but the query model does not accept recursive structures. In this case we will not be able to query for sections included in others, although this may be the case in the texts documents under consideration.

### 4.4.1 Fixed Structure

The structure allowed in texts was traditionally quite restrictive. The documents had a fixed set of *fields*, much like a filled form. Each field had some text inside. Some fields were not present in all documents. Only rarely could the fields appear in any order or repeat across a document. A document could not have text not classified under any field. Fields were not allowed to nest or overlap. The retrieval activity allowed on them was restricted to specifying that a given basic pattern was to be found only in a given field. Most current commercial systems use this model.

This model is reasonable when the text collection has a fixed structure. For instance, a mail archive could be regarded as a set of mails, where each mail has a sender, a receiver, a date, a subject, and a body field. The user can thus search for the mails sent to a given person with '`football`' in the subject field. However, the model is inadequate to represent the hierarchical structure present in an HTML document, for instance.

If the division of the text into fields is rigid enough, the content of some fields can even be interpreted not as text but as numbers, dates, etc. thereby allowing different queries to be posed on them (e.g., month ranges in dates). It is not hard to see that this idea leads naturally to the relational model, each field corresponding to a column in the database table. Looking at the database as a text allows us to query the textual fields with much more power than is common in relational database systems. On the other hand, relational databases may make better use of their knowledge on the data types involved to build specialized and more efficient indices. A number of approaches towards combining these trends have been proposed in recent years, their main problem being that they do not achieve optimal performance because the text is usually stored together with other types of data. Nevertheless, there are several proposals that extend SQL (Structured Query Language) to allow full-text retrieval. Among them we can mention proposals by leading relational database vendors such as Oracle and Sybase, as well as SFQL, which is covered in section 4.5.

### 4.4.2 Hypertext

Hypertexts probably represent the maximum freedom with respect to structuring power. A hypertext is a directed graph where the nodes hold some text and the links represent connections between nodes or between positions inside the nodes (see Chapter 2). Hypertexts have received a lot of attention since the explosion of the Web, which is indeed a gigantic hypertext-like database spread across the world.

However, retrieval from a hypertext began as a merely navigational activity. That is, the user had to manually traverse the hypertext nodes following links to search what he wanted. It was not possible to query the hypertext based on its structure. Even in the Web one can search by the text contents of the nodes, but not by their structural connectivity.

An interesting proposal to combine browsing and searching on the Web is WebGlimpse. It allows classical navigation plus the ability to search by content in the neighborhood of the current node. Currently, some query tools have appeared that achieve the goal of querying hypertexts based on their content and their structure. This problem is covered in detail in Chapter 13.

### 4.4.3 Hierarchical Structure

An intermediate structuring model which lies between fixed structure and hypertext is the hierarchical structure. This model represents a recursive decomposition of the text and is a natural model for many text collections (e.g., books, articles, legal documents, structured programs, etc.). Figure 4.3 shows an example of such a hierarchical structure.

The simplification from hypertext to a hierarchy allows the adoption of faster algorithms to solve queries. As a general rule, the more powerful the model, the less efficiently it can be implemented.

Our aim in this section is to analyze and discuss the different approaches presented by the hierarchical models. We first present a selection of the most representative models and then discuss the main subjects of this area.

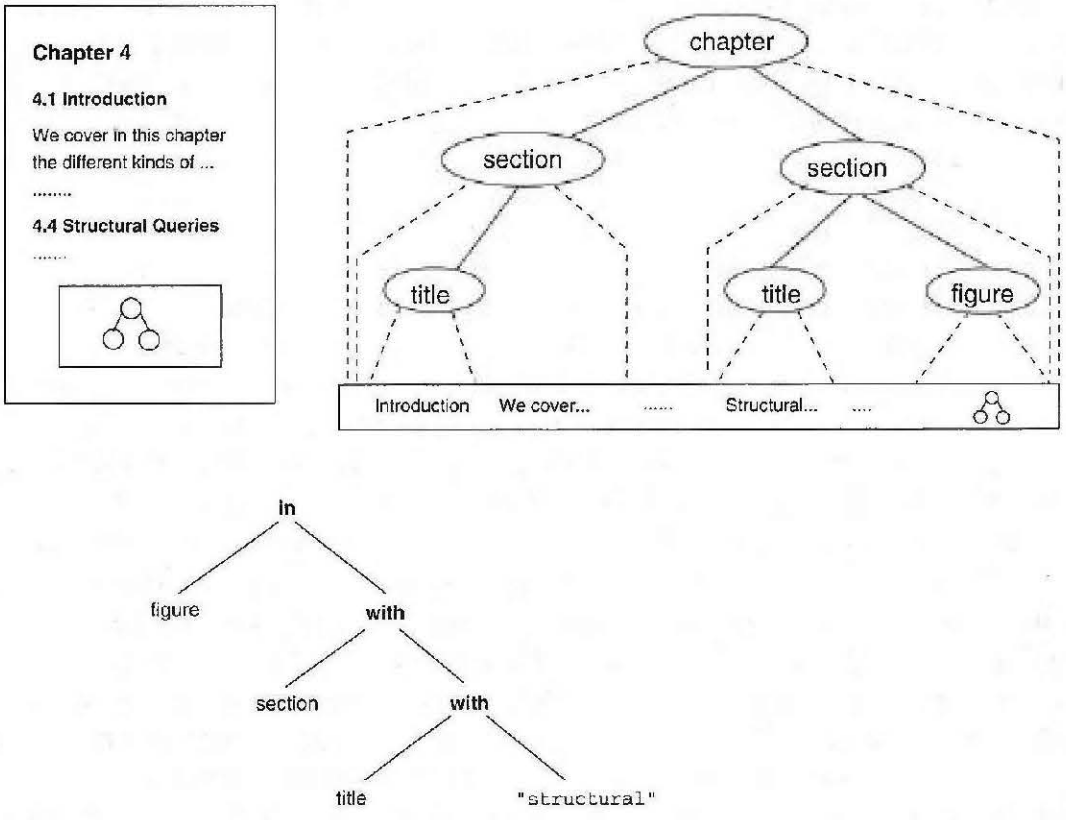



**Figure 4.3** An example of a hierarchical structure: the page of a book, its schematic view, and a parsed query to retrieve the figure.

## A Sample of Hierarchical Models

*PAT Expressions*

These are built on the same index as the text index, i.e. there is no special separate index on the structure. The structure is assumed to be marked in the text by *tags* (as in HTML), and therefore is defined in terms of initial and final tags. This allows a dynamic scheme where the structure of interest is not fixed but can be determined at query time. For instance, since tags need not to be especially designed as normal tags, one can define that the end-of-lines are the marks in order to define a structure on lines. This also allows for a very efficient implementation and no additional space overhead for the structure.

Each pair of initial and final tags defines a *region*, which is a set of contiguous text areas. Externally computed regions are also supported. However, the areas of a region cannot nest or overlap, which is quite restrictive. There is no restriction on areas of different regions.

Apart from text searching operations, it is possible to select areas containing (or not) other areas, contained (or not) in other areas, or followed (or not) by other areas.

A disadvantage is that the algebra mixes regions and sets of text positions which are incompatible and force complex conversion semantics. For instance, if the result of a query is going to generate overlapping areas (a fact that cannot be determined beforehand) then the result is converted to positions. Also, the dynamic definition of regions is flexible but requires the structure to be expressable using tags (also called 'markup', see Chapter 6), which for instance does not occur in some structured programming languages.

*Overlapped Lists*

These can be seen as an evolution of PAT Expressions. The model allows for the areas of a region to overlap, but not to nest. This elegantly solves the problems of mixing regions and sets of positions. The model considers the use of an inverted list (see Chapter 8) where not only the words but also the regions are indexed.

Apart from the operations of PAT Expressions, the model allows us to perform set union, and to combine regions. Combination means selecting the minimal text areas which include any two areas taken from two regions. A 'followed by' operator imposes the additional restriction that the first area must be before the second one. An '$n$ words' operator generates the region of all (overlapping) sequences of $n$ words of the text (this is further used to retrieve elements close to each other). If an operation produces a region with nested areas, only the minimal areas are selected. An example is shown in Figure 2.11.

The implementation of this model can also be very efficient. It is not clear, however, whether overlapping is good or not for capturing the structural properties that information has in practice. A new proposal allows the structure to be nested *and* overlapped, showing that more interesting operators can still be implemented.

*Lists of References*

These are an attempt to make the definition and querying of structured text uniform, using a common language. The language goes beyond querying structured text, so we restrict our attention to the subset in which we are interested.

The structure of documents is fixed and hierarchical, which makes it impossible to have overlapping results. All possible regions are defined at indexing time. The answers delivered are more restrictive, since nesting is not allowed (only the top-level elements qualify) and all elements must be of the same type, e.g. only sections, or only paragraphs. In fact, there are also hypertext links but these cannot be queried (the model also has navigational features).

A static hierarchical structure makes it possible to speak in terms of direct ancestry of nodes, a concept difficult to express when the structure is dynamic. The language allows for querying on 'path expressions,' which describe paths in the structure tree.

Answers to queries are seen as lists of 'references.' A reference is a pointer to a region of the database. This integrates in an elegant way answers to queries and hypertext links, since all are lists of references.

*Proximal Nodes*

This model tries to find a good compromise between expressiveness and efficiency. It does not define a specific language, but a model in which it is shown that a number of useful operators can be included achieving good efficiency.

The structure is fixed and hierarchical. However, many independent structures can be defined on the same text, each one being a strict hierarchy but allowing overlaps between areas of different hierarchies. An example is shown in Figure 2.12.

A query can relate different hierarchies, but returns a subset of the nodes of one hierarchy only (i.e., nested elements are allowed in the answers, but no overlaps). Text matching queries are modeled as returning nodes from a special 'text hierarchy.'

The model specifies a fully compositional language where the leaves of the query syntax tree are formed by basic queries on contents or names of structural elements (e.g., all chapters). The internal nodes combine results. For efficiency, the operations defined at the internal nodes must be implementable looking at the identity and text areas of the operands, and must relate nodes which are close in the text.

It has been shown that many useful operators satisfy this restriction: selecting elements that (directly or transitively) include or are included in others; that are included at a given position (e.g., the third paragraph of each chapter); that are shortly before or after others; set manipulation; and many powerful variations. Operations on content elements deliver a set of regions with no nesting, and those results can be fully integrated into any query. This ability to integrate the text into the model is very useful. On the other hand, some queries requiring non-proximal operations are not allowed, for instance *semijoins*. An example of a semijoin is 'give me the titles of all the chapters referenced in this chapter.'

*Tree Matching*
This model relies on a single primitive: tree inclusion, whose main idea is as follows. Interpreting the structure both of the text database and of the query (which is defined as a pattern on the structure) as trees, determine an embedding of the query into the database which respects the hierarchical relationships between nodes of the query.

Two variants are studied. *Ordered inclusion* forces the embedding to respect the left-to-right relations among siblings in the query, while *unordered inclusion* does not. The leaves of the query can be not only structural elements but also text patterns, meaning that the ancestor of the leaf must contain that pattern.

Simple queries return the roots of the matches. The language is enriched by Prolog-like variables, which can be used to express requirements on equality between parts of the matched substructure and to retrieve another part of the match, not only the root. Logical variables are also used for union and intersection of queries, as well as to emulate tuples and join capabilities.

Although the language is set oriented, the algorithms work by sequentially obtaining each match. The use of logical variables and unordered inclusion makes the search problem intractable (NP-hard in many cases). Even the good cases have an inefficient solution in practice.

## Discussion

A survey of the main hierarchical models raises a number of interesting issues, most of them largely unresolved up to now. Some of them are listed below.

*Static or dynamic structure*
As seen, in a static structure there are one or more explicit hierarchies (which can be queried, e.g., by ancestry), while in a dynamic structure there is not really a hierarchy, but the required elements are built on the fly. A dynamic structure is implemented over a normal text index, while a static one may or may not be. A static structure is independent of the text markup, while a dynamic one is more flexible for building arbitrary structures.

*Restrictions on the structure*
The text or the answers may have restrictions about nesting and/or overlapping. In some cases these restrictions exist for efficiency reasons. In other cases, the query language is restricted to avoid restricting the structure. This choice is largely dependent on the needs of each application.

*Integration with text*
In many structured models, the text content is merely seen as a secondary source of information which is used only to restrict the matches of structural elements. In classic IR models, on the other side, information on the structure is the secondary element which is used only to restrict text matches. For an effective

integration of queries on text content with queries on text structure, the query language must provide for full expressiveness of both types of queries and for effective means of combining them.

*Query language*
Typical queries on structure allow the selection of areas that contain (or not) other areas, that are contained (or not) in other areas, that follow (or are followed by) other areas, that are close to other areas, and set manipulation. Many of them are implemented in most models, although each model has unique features. Some kind of standardization, expressiveness taxonomy, or formal categorization would be highly desirable but does not exist yet.

## 4.5 Query Protocols

In this section we briefly cover some query languages that are used automatically by software applications to query text databases. Some of them are proposed as standards for querying CD-ROMs or as intermediate languages to query library systems. Because they are not intended for human use, we refer to them as protocols rather than languages. More information on protocols can be found in Chapters 14 and 15. The most important query protocols are:

- **Z39.50** is a protocol approved as a standard in 1995 by ANSI and NISO. This protocol is intended to query bibliographical information using a standard interface between the client and the host database manager which is independent of the client user interface and of the query database language at the host. The database is assumed to be a text collection with some fixed fields (although it is more flexible than usual). The Z39.50 protocol is used broadly and is part, for instance, of WAIS (see below). The protocol does not only specify the query language and its semantics, but also the way in which client and server establish a session, communicate and exchange information, etc. Although originally conceived only to operate on bibliographical information (using the Machine Readable Cataloging Record (MARC) format), it has been extended to query other types of information as well.
- **WAIS** (Wide Area Information Service) is a suite of protocols that was popular at the beginning of the 1990s before the boom of the Web. The goal of WAIS was to be a network publishing protocol and to be able to query databases through the Internet.

In the CD-ROM publishing arena, there are several proposals for query protocols. The main goal of these protocols is to provide 'disk interchangeability.' This means more flexibility in data communication between primary information providers and end users. It also enables significant cost savings since it allows access to diverse information without the need to buy, install, and train users for different data retrieval applications. We briefly cover three of these proposals:

- **CCL** (Common Command Language) is a NISO proposal (Z39.58 or ISO 8777) based on Z39.50. It defines 19 commands that can be used interactively. It is more popular in Europe, although very few products use it. It is based on the classical Boolean model.
- **CD-RDx** (Compact Disk Read only Data exchange) uses a client-server architecture and has been implemented in most platforms. The client is generic while the server is designed and provided by the CD-ROM publisher who includes it with the database in the CD-ROM. It allows fixed-length fields, images, and audio, and is supported by such US national agencies as the CIA, NASA, and GSA.
- **SFQL** (Structured Full-text Query Language) is based on SQL and also has a client-server architecture. SFQL has been adopted as a standard by the aerospace community (the Air Transport Association/Aircraft Industry Association). Documents are rows in a relational table and can be tagged using SGML. The language defines the format of the answer, which has a header and a variable length message area. The language does not define any specific formatting or markup. For example, a query in SFQL is:

  ```
  Select abstract from journal.papers where title
  contains "text search"
  ```

  The language supports Boolean and logical operators, thesaurus, proximity operations, and some special characters such as wild cards and repetition. For example:

  ```
  ... where paper contains "retrieval" or like "info
  %" and date > 1/1/98
  ```

  Compared with CCL or CD-RDx, SFQL is more general and flexible, although it is based on a relational model, which is not always the best choice for a document database.

## 4.6 Trends and Research Issues

We reviewed in this chapter the main aspects of the query languages that retrieve information from textual databases. Our discussion covered from the most classic tools to the most novel capabilities that are emerging, from searching words to extended patterns, from the Boolean model to querying structures. Table 4.1 shows the different basic queries allowed in the different models. Although the probabilistic and the Bayesian belief network (BBN) models are based on word queries, they can incorporate set operations.

We present in Figure 4.4 the types of operations we covered and how they can be structured (not all of them exist in all models and not all of them have to be used to form a query). The figure shows, for instance, that we can form a query using Boolean operations over phrases (skipping structural queries), which

| Model | Queries allowed |
|---|---|
| Boolean | word, set operations |
| Vector | words |
| Probabilistic | words |
| BBN | words |

**Table 4.1** Relationship between types of queries and models.

can be formed by words and by regular expressions (skipping the ability to allow errors).

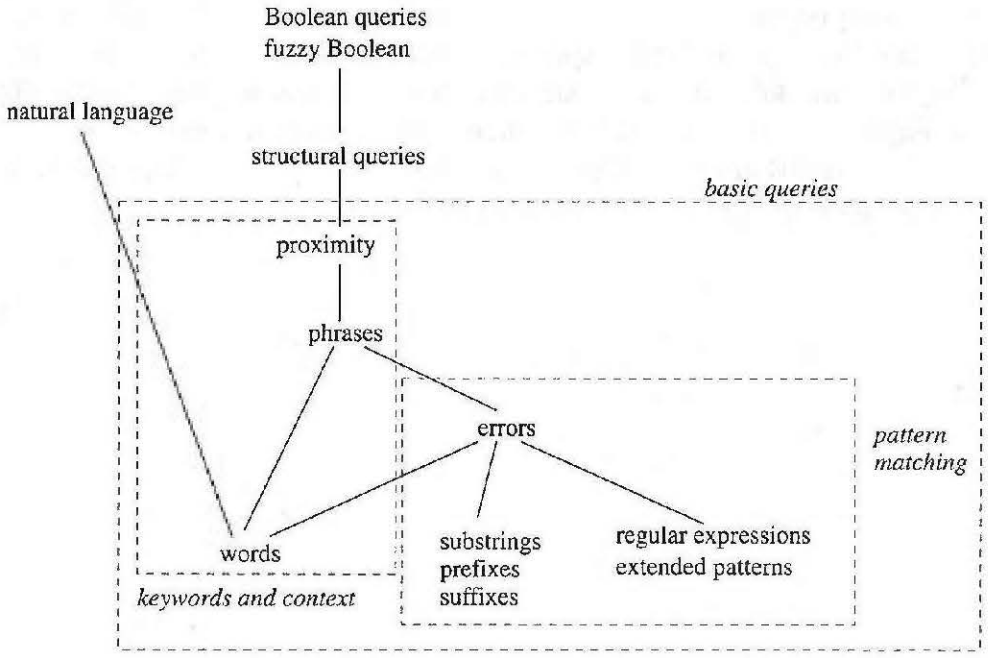



**Figure 4.4** The types of queries covered and how they are structured.

The area of query languages for text databases is definitely moving towards higher flexibility. While text models are moving towards the goal of achieving a better understanding of the user needs (by providing relevance feedback, for instance), the query languages are allowing more and more power in the specification of the query. While extended patterns and searching allowing errors permit us to find patterns without complete knowledge of what is wanted, querying on the structure of the text (and not only on its content) provides greater expressiveness and increased functionality.

Another important research topic is visual query languages. Visual metaphors can help non-experienced users to pose complex Boolean queries. Also, a visual query language can include the structure of the document. This topic is related to user interfaces and visualization and is covered in Chapter 10.

## 4.7 Bibliographic Discussion

The material on classical query languages (most simple patterns, Boolean model, and fixed structure) is based on current commercial systems, such as Fulcrum, Verity, and others, as well as on non-commercial systems such as Glimpse [540] and Igrep [26].

The fuzzy Boolean model is described in [703]. The Levenshtein distance is described in [504] and [25]. Soundex is explained in [445]. A comparison of the effectiveness of different similarity models is given in [595]. A good source on regular expressions is [375]. A rich language on extended patterns is described in [837].

A classical reference on hypertext is [181]. The WebGlimpse system is presented in [539]. The discussion of hierarchical text is partially based on [41]. The original proposals are: PAT Expressions [693], Overlapped Lists [173] and the new improved proposal [206], Lists of References [534], Proximal Nodes [590], and Tree Matching [439]. PAT Expressions are the basic model of the PAT Text Searching System [309]. A simple structured text model is presented in [36] and a visual query language that includes structure is discussed in [44].

More information on Z39.50 can be obtained from [23]. More information on WAIS is given in [425]. For details on SFQL see [392].

# Chapter 5
# Query Operations

## 5.1 Introduction

Without detailed knowledge of the collection make-up and of the retrieval environment, most users find it difficult to formulate queries which are well designed for retrieval purposes. In fact, as observed with Web search engines, the users might need to spend large amounts of time reformulating their queries to accomplish effective retrieval. This difficulty suggests that the first query formulation should be treated as an initial (naive) attempt to retrieve relevant information. Following that, the documents initially retrieved could be examined for relevance and new improved query formulations could then be constructed in the hope of retrieving additional useful documents. Such query reformulation involves two basic steps: expanding the original query with new terms and reweighting the terms in the expanded query.

In this chapter, we examine a variety of approaches for improving the initial query formulation through query expansion and term reweighting. These approaches are grouped in three categories: (a) approaches based on feedback information from the user; (b) approaches based on information derived from the set of documents initially retrieved (called the *local* set of documents); and (c) approaches based on global information derived from the document collection. In the first category, user relevance feedback methods for the vector and probabilistic models are discussed. In the second category, two approaches for local analysis (i.e., analysis based on the set of documents initially retrieved) are presented. In the third category, two approaches for global analysis are covered.

Our discussion is not aimed at completely covering the area, neither does it intend to present an exhaustive survey of query operations. Instead, our discussion is based on a selected bibliography which, we believe, is broad enough to allow an overview of the main issues and tradeoffs involved in query operations. Local and global analysis are highly dependent on clustering algorithms. Thus, clustering is covered throughout our discussion. However, there is no intention of providing a complete survey of clustering algorithms for information retrieval.

## 5.2 User Relevance Feedback

*Relevance feedback* is the most popular query reformulation strategy. In a relevance feedback cycle, the user is presented with a list of the retrieved documents and, after examining them, marks those which are relevant. In practice, only the top 10 (or 20) ranked documents need to be examined. The main idea consists of selecting important terms, or expressions, attached to the documents that have been identified as relevant by the user, and of enhancing the importance of these terms in a new query formulation. The expected effect is that the new query will be moved towards the relevant documents and away from the non-relevant ones.

Early experiments using the Smart system [695] and later experiments using the probabilistic weighting model [677] have shown good improvements in precision for small test collections when relevance feedback is used. Such improvements come from the use of two basic techniques: query expansion (addition of new terms from relevant documents) and term reweighting (modification of term weights based on the user relevance judgement).

Relevance feedback presents the following main advantages over other query reformulation strategies: (a) it shields the user from the details of the query reformulation process because all the user has to provide is a relevance judgement on documents; (b) it breaks down the whole searching task into a sequence of small steps which are easier to grasp; and (c) it provides a controlled process designed to emphasize some terms (relevant ones) and de-emphasize others (non-relevant ones).

In the following three subsections, we discuss the usage of user relevance feedback to (a) expand queries with the vector model, (b) reweight query terms with the probabilistic model, and (c) reweight query terms with a variant of the probabilistic model.

### 5.2.1 Query Expansion and Term Reweighting for the Vector Model

The application of *relevance feedback* to the vector model considers that the term-weight vectors of the documents identified as relevant (to a given query) have similarities among themselves (i.e., relevant documents resemble each other). Further, it is assumed that non-relevant documents have term-weight vectors which are dissimilar from the ones for the relevant documents. The basic idea is to reformulate the query such that it gets closer to the term-weight vector space of the relevant documents.

Let us define some additional terminology regarding the processing of a given query $q$ as follows,

$D_r$: set of relevant documents, as identified by the user, among the *retrieved* documents;
$D_n$: set of non-relevant documents among the *retrieved* documents;
$C_r$: set of relevant documents among all documents in the collection;

$|D_r|, |D_n|, |C_r|$: number of documents in the sets $D_r$, $D_n$, and $C_r$, respectively;
$\alpha, \beta, \gamma$: tuning constants.

Consider first the unrealistic situation in which the complete set $C_r$ of relevant documents to a given query $q$ is known in advance. In such a situation, it can be demonstrated that the best query vector for distinguishing the relevant documents from the non-relevant documents is given by,

$$\vec{q}_{opt} = \frac{1}{|C_r|} \sum_{\forall \vec{d}_j \in C_r} \vec{d}_j - \frac{1}{N - |C_r|} \sum_{\forall \vec{d}_j \notin C_r} \vec{d}_j \tag{5.1}$$

The problem with this formulation is that the *relevant* documents which compose the set $C_r$ are not known a priori. In fact, we are looking for them. The natural way to avoid this problem is to formulate an initial query and to incrementally change the initial query vector. This incremental change is accomplished by restricting the computation to the documents *known* to be relevant (according to the user judgement) at that point. There are three classic and similar ways to calculate the modified query $\vec{q}_m$ as follows,

$$\textit{Standard\_Rochio}: \quad \vec{q}_m = \alpha\, \vec{q} + \frac{\beta}{|D_r|} \sum_{\forall \vec{d}_j \in D_r} \vec{d}_j - \frac{\gamma}{|D_n|} \sum_{\forall \vec{d}_j \in D_n} \vec{d}_j$$

$$\textit{Ide\_Regular}: \quad \vec{q}_m = \alpha\, \vec{q} + \beta \sum_{\forall \vec{d}_j \in D_r} \vec{d}_j - \gamma \sum_{\forall \vec{d}_j \in D_n} \vec{d}_j$$

$$\textit{Ide\_Dec\_Hi}: \quad \vec{q}_m = \alpha\, \vec{q} + \beta \sum_{\forall \vec{d}_j \in D_r} \vec{d}_j - \gamma\ max_{non\text{-}relevant}(\vec{d}_j)$$

where $max_{non-relevant}(\vec{d}_j)$ is a reference to the highest ranked non-relevant document. Notice that now $D_r$ and $D_n$ stand for the sets of relevant and non-relevant documents (among the retrieved ones) according to the user judgement, respectively. In the original formulations, Rochio [678] fixed $\alpha = 1$ and Ide [391] fixed $\alpha = \beta = \gamma = 1$. The expressions above are modern variants. The current understanding is that the three techniques yield similar results (in the past, Ide Dec-Hi was considered slightly better).

The Rochio formulation is basically a direct adaptation of equation 5.1 in which the terms of the original query are added in. The motivation is that in practice the original query $q$ may contain important information. Usually, the information contained in the relevant documents is more important than the information provided by the non-relevant documents [698]. This suggests making the constant $\gamma$ smaller than the constant $\beta$. An alternative approach is to set $\gamma$ to 0 which yields a *positive* feedback strategy.

The main advantages of the above relevance feedback techniques are simplicity and good results. The simplicity is due to the fact that the modified term weights are computed directly from the set of retrieved documents. The good

results are observed experimentally and are due to the fact that the modified query vector does reflect a portion of the intended query semantics. The main disadvantage is that *no* optimality criterion is adopted.

### 5.2.2 Term Reweighting for the Probabilistic Model

The probabilistic model dynamically ranks documents similar to a query $q$ according to the probabilistic ranking principle. From Chapter 2, we already know that the similarity of a document $d_j$ to a query $q$ can be expressed as

$$sim(d_j, q) \propto \sum_{i=1}^{t} w_{i,q}\, w_{i,j} \left( \log \frac{P(k_i|R)}{1 - P(k_i|R)} + \log \frac{1 - P(k_i|\overline{R})}{P(k_i|\overline{R})} \right) \qquad (5.2)$$

where $P(k_i|R)$ stands for the probability of observing the term $k_i$ in the set $R$ of relevant documents and $P(k_i|\overline{R})$ stands for the probability of observing the term $k_i$ in the set $\overline{R}$ of non-relevant documents.

Initially, equation 5.2 cannot be used because the probabilities $P(k_i|R)$ and $P(k_i|\overline{R})$ are unknown. A number of different methods for estimating these probabilities automatically (i.e., without feedback from the user) were discussed in Chapter 2. With user feedback information, these probabilities are estimated in a slightly different way as follows.

For the initial search (when there are no retrieved documents yet), assumptions often made include: (a) $P(k_i|R)$ is constant for all terms $k_i$ (typically 0.5) and (b) the term probability distribution $P(k_i|\overline{R})$ can be approximated by the distribution in the whole collection. These two assumptions yield:

$$\begin{aligned} P(k_i|R) &= 0.5 \\ P(k_i|\overline{R}) &= \frac{n_i}{N} \end{aligned}$$

where, as before, $n_i$ stands for the number of documents in the collection which contain the term $k_i$. Substituting into equation 5.2, we obtain

$$sim_{initial}(d_j, q) = \sum_{i}^{t} w_{i,q}\, w_{i,j}\, \log \frac{N - n_i}{n_i}$$

For the feedback searches, the accumulated statistics related to the relevance or non-relevance of previously retrieved documents are used to evaluate the probabilities $P(k_i|R)$ and $P(k_i|\overline{R})$. As before, let $D_r$ be the set of relevant retrieved documents (according to the user judgement) and $D_{r,i}$ be the subset of $D_r$ composed of the documents which contain the term $k_i$. Then, the probabilities $P(k_i|R)$ and $P(k_i|\overline{R})$ can be approximated by

$$P(k_i|R) = \frac{|D_{r,i}|}{|D_r|}; \quad P(k_i|\overline{R}) = \frac{n_i - |D_{r,i}|}{N - |D_r|} \qquad (5.3)$$

Using these approximations, equation 5.2 can rewritten as

$$sim(d_j, q) = \sum_{i=1}^{t} w_{i,q}\ w_{i,j}\ \log\left[\frac{|D_{r,i}|}{|D_r| - |D_{r,i}|} \div \frac{n_i - |D_{r,i}|}{N - |D_r| - (n_i - |D_{r,i}|)}\right]$$

Notice that here, contrary to the procedure in the vector space model, no query expansion occurs. The same query terms are being reweighted using feedback information provided by the user.

Formula 5.3 poses problems for certain small values of $|D_r|$ and $|D_{r,i}|$ that frequently arise in practice ($|D_r| = 1, |D_{r,i}| = 0$). For this reason, a 0.5 adjustment factor is often added to the estimation of $P(k_i|R)$ and $P(k_i|\overline{R})$ yielding

$$P(k_i|R) = \frac{|D_{r,i}| + 0.5}{|D_r| + 1}; \quad P(k_i|\overline{R}) = \frac{n_i - |D_{r,i}| + 0.5}{N - |D_r| + 1} \tag{5.4}$$

This 0.5 adjustment factor may provide unsatisfactory estimates in some cases, and alternative adjustments have been proposed such as $n_i/N$ or $(n_i - |D_{r,i}|)/(N - |D_r|)$ [843]. Taking $n_i/N$ as the adjustment factor (instead of 0.5), equation 5.4 becomes

$$P(k_i|R) = \frac{|D_{r,i}| + \frac{n_i}{N}}{|D_r| + 1}; \quad P(k_i|\overline{R}) = \frac{n_i - |D_{r,i}| + \frac{n_i}{N}}{N - |D_r| + 1}$$

The main advantages of this relevance feedback procedure are that the feedback process is directly related to the derivation of new weights for *query* terms and that the term reweighting is optimal under the assumptions of term independence and binary document indexing ($w_{i,q} \in \{0, 1\}$ and $w_{i,j} \in \{0, 1\}$). The disadvantages include: (1) document term weights are *not* taken into account during the feedback loop; (2) weights of terms in the previous query formulations are also disregarded; and (3) no query expansion is used (the same set of index terms in the original query is reweighted over and over again). As a result of these disadvantages, the probabilistic relevance feedback methods do *not* in general operate as effectively as the conventional vector modification methods.

To extend the probabilistic model with query expansion capabilities, different approaches have been proposed in the literature ranging from term weighting for query expansion to term clustering techniques based on spanning trees. All of these approaches treat probabilistic query expansion separately from probabilistic term reweighting. While we do not discuss them here, a brief history of research on this issue and bibliographical references can be found in section 5.6.

### 5.2.3 A Variant of Probabilistic Term Reweighting

The discussion above on term reweighting is based on the classic probabilistic model introduced by Robertson and Sparck Jones in 1976. In 1983, Croft extended this weighting scheme by suggesting distinct initial search methods

and by adapting the probabilistic formula to include within-document frequency weights. This variant of probabilistic term reweighting is more flexible (and also more powerful) and is briefly reviewed in this section.

The formula 5.2 for probabilistic ranking can be rewritten as

$$sim(d_j, q) \; \alpha \; \sum_{i=1}^{t} w_{i,q} \; w_{i,j} \; F_{i,j,q}$$

where $F_{i,j,q}$ is interpreted as a factor which depends on the triple $[k_i, d_j, q]$. In the classic formulation, $F_{i,j,q}$ is computed as a function of $P(k_i|R)$ and $P(k_i|\overline{R})$ (see equation 5.2). In his variant, Croft proposed that the initial search and the feedback searches use distinct formulations.

For the initial search, he suggested

$$\begin{aligned} F_{i,j,q} &= (C + idf_i) \; \overline{f}_{i,j} \\ \overline{f}_{i,j} &= K + (1 + K) \; \frac{f_{i,j}}{max(f_{i,j})} \end{aligned}$$

where $\overline{f}_{i,j}$ is a normalized within-document frequency. The parameters $C$ and $K$ should be adjusted according to the collection. For automatically indexed collections, $C$ should be initially set to 0.

For the feedback searches, Croft suggested the following formulation for $F_{i,j,q}$

$$F_{i,j,q} = \left( C + \log \frac{P(k_i|R)}{1 - P(k_i|R)} + \log \frac{1 - P(k_i|\overline{R})}{P(k_i|\overline{R})} \right) \; \overline{f}_{i,j}$$

where $P(k_i|R)$ and $P(k_i|\overline{R})$ are computed as in equation 5.4.

This variant of probabilistic term reweighting has the following advantages: (1) it takes into account the within-document frequencies; (2) it adopts a normalized version of these frequencies; and (3) it introduces the constants $C$ and $K$ which provide for greater flexibility. However, it constitutes a more complex formulation and, as before, it operates solely on the terms originally in the query (without query expansion).

### 5.2.4 Evaluation of Relevance Feedback Strategies

Consider the modified query vector $\vec{q}_m$ generated by the Rochio formula and assume that we want to evaluate its retrieval performance. A simplistic approach is to retrieve a set of documents using $\vec{q}_m$, to rank them using the vector formula, and to measure recall-precision figures relative to the set of relevant documents (provided by the experts) for the original query vector $\vec{q}$. In general, the results show spectacular improvements. Unfortunately, a significant part of this improvement results from the higher ranks assigned to the set $R$ of documents

already identified as relevant during the feedback process [275]. Since the user has seen these documents already (and pointed them as relevants), such evaluation is unrealistic. Further, it masks any real gains in retrieval performance due to documents not seen by the user yet.

A more realistic approach is to evaluate the retrieval performance of the modified query vector $\vec{q}_m$ considering only the *residual collection* i.e., the set of all documents minus the set of feedback documents provided by the user. Because highly ranked documents are removed from the collection, the recall-precision figures for $\vec{q}_m$ tend to be lower than the figures for the original query vector $\vec{q}$. This is not a limitation because our main purpose is to compare the performance of distinct relevance feedback strategies (and not to compare the performance before and after feedback). Thus, as a basic rule of thumb, any experimentation involving relevance feedback strategies should always evaluate recall-precision figures relative to the residual collection.

## 5.3 Automatic Local Analysis

In a user relevance feedback cycle, the user examines the top ranked documents and separates them into two classes: the relevant ones and the non-relevant ones. This information is then used to select new terms for query expansion. The reasoning is that the expanded query will retrieve more relevant documents. Thus, there is an underlying notion of *clustering* supporting the feedback strategy. According to this notion, known relevant documents contain terms which can be used to describe a larger cluster of relevant documents. In this case, the description of this larger cluster of relevant documents is built interactively with assistance from the user.

A distinct approach is to attempt to obtain a description for a larger cluster of relevant documents automatically. This usually involves identifying terms which are related to the query terms. Such terms might be synonyms, stemming variations, or terms which are close to the query terms in the text (i.e., terms with a distance of at most $k$ words from a query term). Two basic types of strategies can be attempted: global ones and local ones.

In a global strategy, all documents in the collection are used to determine a global thesaurus-like structure which defines term relationships. This structure is then shown to the user who selects terms for query expansion. Global strategies are discussed in section 5.4.

In a local strategy, the documents retrieved for a given query $q$ are examined at query time to determine terms for query expansion. This is similar to a relevance feedback cycle but might be done without assistance from the user (i.e., the approach might be fully automatic). Two local strategies are discussed below: local clustering and local context analysis. The first is based on the work done by Attar and Fraenkel in 1977 and is used here to establish many of the fundamental ideas and concepts regarding the usage of clustering for query expansion. The second is a recent work done by Xu and Croft in 1996 and illustrates the advantages of combining techniques from both local and global analysis.

### 5.3.1 Query Expansion Through Local Clustering

Adoption of clustering techniques for query expansion is a basic approach which has been attempted since the early years of information retrieval. The standard approach is to build global structures such as association matrices which quantify term correlations (for instance, number of documents in which two given terms co-occur) and to use correlated terms for query expansion. The main problem with this strategy is that there is not consistent evidence that global structures can be used effectively to improve retrieval performance with general collections. One main reason seems to be that global structures do not adapt well to the local context defined by the current query. One approach to deal with this effect is to devise strategies which aim at optimizing the current search. Such strategies are based on *local clustering* and are now discussed. Our discussion is based on the original work by Attar and Fraenkel which appeared in 1977.

We first define basic terminology as follows.

**Definition** *Let $V(s)$ be a non-empty subset of words which are grammatical variants of each other. A canonical form $s$ of $V(s)$ is called a* stem. *For instance, if $V(s)$=\{polish,polishing,polished\} then $s$=polish.*

For a detailed discussion on stemming algorithms see Chapter 7. While stems are adopted in our discussion, the ideas below are also valid for non-stemmed keywords. We proceed with a characterization of the local nature of the strategies covered here.

**Definition** *For a given query $q$, the set $D_l$ of documents retrieved is called the local document set. Further, the set $V_l$ of all distinct words in the local document set is called the local vocabulary. The set of all distinct stems derived from the set $V_l$ is referred to as $S_l$.*

We operate solely on the documents retrieved for the current query. Since it is frequently necessary to access the text of such documents, the application of local strategies to the Web is unlikely at this time. In fact, at a client machine, retrieving the text of 100 Web documents for local analysis would take too long, reducing drastically the interactive nature of Web interfaces and the satisfaction of the users. Further, at the search engine site, analyzing the text of 100 Web documents would represent an extra spending of CPU time which is not cost effective at this time (because search engines depend on processing a high number of queries per unit of time for economic survival). However, local strategies might be quite useful in the environment of intranets such as, for instance, the collection of documents issued by a large business company. Further, local strategies might also be of great assistance for searching information in specialized document collections (for instance, medical document collections).

Local feedback strategies are based on expanding the query with terms correlated to the query terms. Such correlated terms are those present in local clusters built from the local document set. Thus, before we discuss local

query expansion, we discuss strategies for building local clusters. Three types of clusters are covered: association clusters, metric clusters, and scalar clusters.

### Association Clusters

An association cluster is based on the co-occurrence of stems (or terms) inside documents. The idea is that stems which co-occur frequently inside documents have a synonymity association. Association clusters are generated as follows.

**Definition** *The frequency of a stem $s_i$ in a document $d_j$, $d_j \in D_l$, is referred to as $f_{s_i,j}$. Let $\vec{m}=(m_{ij})$ be an association matrix with $|S_l|$ rows and $|D_l|$ columns, where $m_{ij}=f_{s_i,j}$. Let $\vec{m}^t$ be the transpose of $\vec{m}$. The matrix $\vec{s}=\vec{m}\vec{m}^t$ is a local stem-stem association matrix. Each element $s_{u,v}$ in $\vec{s}$ expresses a correlation $c_{u,v}$ between the stems $s_u$ and $s_v$ namely,*

$$c_{u,v} = \sum_{d_j \in D_l} f_{s_u,j} \times f_{s_v,j} \tag{5.5}$$

The correlation factor $c_{u,v}$ quantifies the absolute frequencies of co-occurrence and is said to be unnormalized. Thus, if we adopt

$$s_{u,v} = c_{u,v} \tag{5.6}$$

then the association matrix $\vec{s}$ is said to be unnormalized. An alternative is to normalize the correlation factor. For instance, if we adopt

$$s_{u,v} = \frac{c_{u,v}}{c_{u,u} + c_{v,v} - c_{u,v}} \tag{5.7}$$

then the association matrix $\vec{s}$ is said to be normalized. The adoption of normalization yields quite distinct associations as discussed below.

Given a local association matrix $\vec{s}$, we can use it to build local association clusters as follows.

**Definition** *Consider the u-th row in the association matrix $\vec{s}$ (i.e., the row with all the associations for the stem $s_u$). Let $S_u(n)$ be a function which takes the u-th row and returns the set of n largest values $s_{u,v}$, where v varies over the set of local stems and $v \neq u$. Then $S_u(n)$ defines a local association cluster around the stem $s_u$. If $s_{u,v}$ is given by equation 5.6, the association cluster is said to be unnormalized. If $s_{u,v}$ is given by equation 5.7, the association cluster is said to be normalized.*

Given a query $q$, we are normally interested in finding clusters only for the $|q|$ query terms. Further, it is desirable to keep the size of such clusters small. This means that such clusters can be computed efficiently at query time.

Despite the fact that the above clustering procedure adopts stems, it can equally be applied to non-stemmed keywords. The procedure remains unchanged except for the usage of keywords instead of stems. Keyword-based local clustering is equally worthwhile trying because there is controversy over the advantages of using a stemmed vocabulary, as discussed in Chapter 7.

### Metric Clusters

Association clusters are based on the frequency of co-occurrence of pairs of terms in documents and do not take into account *where* the terms occur in a document. Since two terms which occur in the same sentence seem more correlated than two terms which occur far apart in a document, it might be worthwhile to factor in the distance between two terms in the computation of their correlation factor. Metric clusters are based on this idea.

**Definition** *Let the distance $r(k_i, k_j)$ between two keywords $k_i$ and $k_j$ be given by the number of words between them in a same document. If $k_i$ and $k_j$ are in distinct documents we take $r(k_i, k_j) = \infty$. A local stem-stem metric correlation matrix $\vec{s}$ is defined as follows. Each element $s_{u,v}$ of $\vec{s}$ expresses a metric correlation $c_{u,v}$ between the stems $s_u$ and $s_v$ namely,*

$$c_{u,v} = \sum_{k_i \in V(s_u)} \sum_{k_j \in V(s_v)} \frac{1}{r(k_i, k_j)}$$

In this expression, as already defined, $V(s_u)$ and $V(s_v)$ indicate the sets of keywords which have $s_u$ and $s_v$ as their respective stems. Variations of the above expression for $c_{u,v}$ have been reported in the literature (such as $1/r^2(k_i, k_j)$) but the differences in experimental results are not remarkable.

The correlation factor $c_{u,v}$ quantifies absolute inverse distances and is said to be unnormalized. Thus, if we adopt

$$s_{u,v} = c_{u,v} \tag{5.8}$$

then the association matrix $\vec{s}$ is said to be unnormalized. An alternative is to normalize the correlation factor. For instance, if we adopt

$$s_{u,v} = \frac{c_{u,v}}{|V(s_u)| \times |V(s_v)|} \tag{5.9}$$

then the association matrix $\vec{s}$ is said to be normalized.

Given a local metric matrix $\vec{s}$, we can use it to build local metric clusters as follows.

**Definition** *Consider the u-th row in the metric correlation matrix $\vec{s}$ (i.e., the row with all the associations for the stem $s_u$). Let $S_u(n)$ be a function which*

*takes the u-th row and returns the set of n largest values $s_{u,v}$, where v varies over the set of local stems and $v \neq u$. Then $S_u(n)$ defines a local metric cluster around the stem $s_u$. If $s_{u,v}$ is given by equation 5.8, the metric cluster is said to be unnormalized. If $s_{u,v}$ is given by equation 5.9, the metric cluster is said to be normalized.*

### Scalar Clusters

One additional form of deriving a synonymity relationship between two local stems (or terms) $s_u$ and $s_v$ is by comparing the sets $S_u(n)$ and $S_v(n)$. The idea is that two stems with similar *neighborhoods* have some synonymity relationship. In this case we say that the relationship is indirect or induced by the neighborhood. One way of quantifying such neighborhood relationships is to arrange all correlation values $s_{u,i}$ in a vector $\vec{s}_u$, to arrange all correlation values $s_{v,i}$ in another vector $\vec{s}_v$, and to compare these vectors through a scalar measure. For instance, the cosine of the angle between the two vectors is a popular scalar similarity measure.

**Definition** *Let $\vec{s}_u = (s_{u,1}, s_{u,2}, \ldots, s_{u,n})$ and $\vec{s}_v = (s_{v,1}, s_{v,2}, \ldots, s_{v,n})$ be two vectors of correlation values for the stems $s_u$ and $s_v$. Further, let $\vec{s} = (s_{u,v})$ be a scalar association matrix. Then, each $s_{u,v}$ can be defined as*

$$s_{u,v} = \frac{\vec{s}_u \cdot \vec{s}_v}{|\vec{s}_u| \times |\vec{s}_v|} \tag{5.10}$$

The correlation matrix $\vec{s}$ is said to be induced by the neighborhood. Using it, a scalar cluster is then defined as follows.

**Definition** *Let $S_u(n)$ be a function which returns the set of n largest values $s_{u,v}$, $v \neq u$, defined according to equation 5.10. Then, $S_u(n)$ defines a scalar cluster around the stem $s_u$.*

### Interactive Search Formulation

Stems (or terms) that belong to clusters associated to the query stems (or terms) can be used to expand the original query. Such stems are called neighbors (of the query stems) and are characterized as follows.

A stem $s_u$ which belongs to a cluster (of size $n$) associated to another stem $s_v$ (i.e., $s_u \in S_v(n)$) is said to be a *neighbor* of $s_v$. Sometimes, $s_u$ is also called a *searchonym* of $s_v$ but here we opt for using the terminology *neighbor*. While neighbor stems are said to have a synonymity relationship, they are not necessarily synonyms in the grammatical sense. Often, neighbor stems represent distinct keywords which are though correlated by the current query context. The local aspect of this correlation is reflected in the fact that the documents and stems considered in the correlation matrix are all local (i.e., $d_j \in D_l$, $s_u \in V_l$).

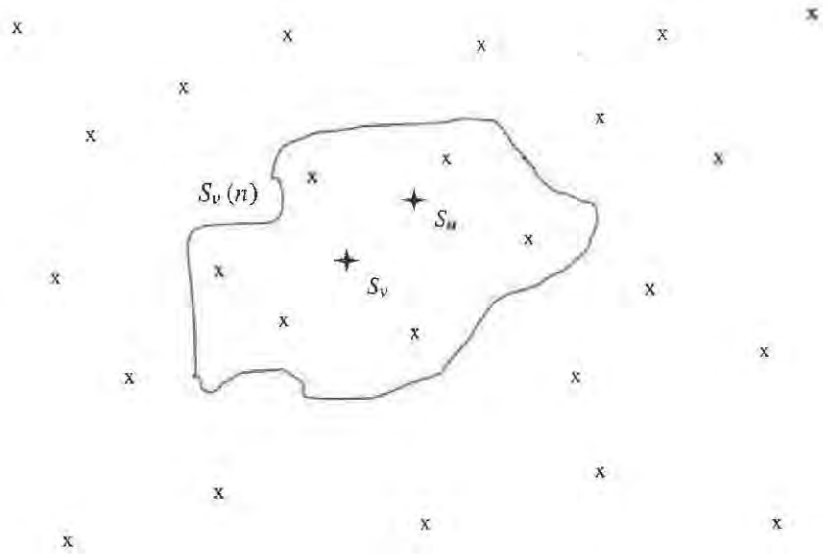



**Figure 5.1** Stem $s_u$ as a neighbor of the stem $s_v$.

Figure 5.1 illustrates a stem (or term) $s_u$ which is located within a neighborhood $S_v(n)$ associated with the stem (or term) $s_v$. In its broad meaning, neighbor stems are an important product of the local clustering process since they can be used for extending a search formulation in a promising unexpected direction, rather than merely complementing it with missing synonyms.

Consider the problem of expanding a given user query $q$ with neighbor stems (or terms). One possibility is to expand the query as follows. For each stem $s_v \in q$, select $m$ neighbor stems from the cluster $S_v(n)$ (which might be of type association, metric, or scalar) and add them to the query. Hopefully, the additional neighbor stems will retrieve new relevant documents. To cover a broader neighborhood, the set $S_v(n)$ might be composed of stems obtained using correlation factors (i.e., $c_{u,v}$) normalized and unnormalized. The qualitative interpretation is that an unnormalized cluster tends to group stems whose ties are due to their large frequencies, while a normalized cluster tends to group stems which are more rare. Thus, the union of the two clusters provides a better representation of the possible correlations.

Besides the merging of normalized and unnormalized clusters, one can also use information about correlated stems to improve the search. For instance, as before, let two stems $s_u$ and $s_v$ be correlated with a correlation factor $c_{u,v}$. If $c_{u,v}$ is larger than a predefined threshold then a neighbor stem of $s_u$ can also be interpreted as a neighbor stem of $s_v$ and vice versa. This provides greater flexibility, particularly with Boolean queries. To illustrate, consider the expression $(s_u + s_v)$ where the + symbol stands for disjunction. Let $s_{u'}$ be a neighbor stem of $s_u$. Then, one can try both $(s_{u'} + s_v)$ and $(s_u + s_{u'})$ as synonym search expressions, because of the correlation given by $c_{u,v}$.

Experimental results reported in the literature usually support the hypothesis of the usefulness of local clustering methods. Furthermore, metric clusters seem to perform better than purely association clusters. This strengthens the hypothesis that there is a correlation between the association of two terms and the distance between them.

We emphasize that all the qualitative arguments in this section are explicitly based on the fact that all the clusters are local (i.e., derived solely from the documents retrieved for the current query). In a global context, clusters are derived from all the documents in the collection which implies that our qualitative argumentation might not stand. The main reason is that correlations valid in the whole corpora might not be valid for the current query.

### 5.3.2 Query Expansion Through Local Context Analysis

The local clustering techniques discussed above are based on the set of documents retrieved for the original query and use the top ranked documents for clustering neighbor terms (or stems). Such a clustering is based on term (stems were considered above) co-occurrence inside documents. Terms which are the best neighbors of each query term are then used to expand the original query $q$. A distinct approach is to search for term correlations in the whole collection — an approach called global analysis. Global techniques usually involve the building of a thesaurus which identifies term relationships in the whole collection. The terms are treated as concepts and the thesaurus is viewed as a concept relationship structure. Thesauri are expensive to build but, besides providing support for query expansion, are useful as a browsing tool as demonstrated by some search engines in the Web. The building of a thesaurus usually considers the use of small contexts and phrase structures instead of simply adopting the context provided by a whole document. Furthermore, with modern variants of global analysis, terms which are closest to the whole query (and not to individual query terms) are selected for query expansion. The application of ideas from global analysis (such as small contexts and phrase structures) to the local set of documents retrieved is a recent idea which we now discuss.

*Local context analysis* [838] combines global and local analysis and works as follows. First, the approach is based on the use of noun groups (i.e., a single noun, two adjacent nouns, or three adjacent nouns in the text), instead of simple keywords, as document concepts. For query expansion, concepts are selected from the top ranked documents (as in local analysis) based on their co-occurrence with query terms (no stemming). However, instead of documents, passages (i.e., a text window of fixed size) are used for determining co-occurrence (as in global analysis).

More specifically, the local context analysis procedure operates in three steps.

- First, retrieve the top $n$ ranked passages using the original query. This is accomplished by breaking up the documents initially retrieved by the

query in fixed length passages (for instance, of size 300 words) and ranking these passages as if they were documents.

- Second, for each concept $c$ in the top ranked passages, the similarity $sim(q, c)$ between the whole query $q$ (not individual query terms) and the concept $c$ is computed using a variant of tf-idf ranking.
- Third, the top $m$ ranked concepts (according to $sim(q, c)$) are added to the original query $q$. To each added concept is assigned a weight given by $1 - 0.9 \times i/m$ where $i$ is the position of the concept in the final concept ranking. The terms in the original query $q$ might be stressed by assigning a weight equal to 2 to each of them.

Of these three steps, the second one is the most complex and the one which we now discuss.

The similarity $sim(q, c)$ between each related concept $c$ and the original query $q$ is computed as follows.

$$sim(q, c) = \prod_{k_i \in q} \left( \delta + \frac{\log(f(c, k_i) \times idf_c)}{\log n} \right)^{idf_i}$$

where $n$ is the number of top ranked passages considered. The function $f(c, k_i)$ quantifies the correlation between the concept $c$ and the query term $k_i$ and is given by

$$f(c, k_i) = \sum_{j=1}^{n} pf_{i,j} \times pf_{c,j}$$

where $pf_{i,j}$ is the frequency of term $k_i$ in the $j$-th passage and $pf_{c,j}$ is the frequency of the concept $c$ in the $j$-th passage. Notice that this is the standard correlation measure defined for association clusters (by Equation 5.5) but adapted for passages. The inverse document frequency factors are computed as

$$idf_i = max(1, \frac{\log_{10} N/np_i}{5})$$
$$idf_c = max(1, \frac{\log_{10} N/np_c}{5})$$

where $N$ is the number of passages in the collection, $np_i$ is the number of passages containing the term $k_i$, and $np_c$ is the number of passages containing the concept $c$. The factor $\delta$ is a constant parameter which avoids a value equal to zero for $sim(q, c)$ (which is useful, for instance, if the approach is to be used with probabilistic frameworks such as that provided by belief networks). Usually, $\delta$ is a small factor with values close to 0.1 (10% of the maximum of 1). Finally, the $idf_i$ factor in the exponent is introduced to emphasize infrequent query terms.

The procedure above for computing $sim(q, c)$ is a non-trivial variant of tf-idf ranking. Furthermore, it has been adjusted for operation with TREC data

and did not work so well with a different collection. Thus, it is important to have in mind that tuning might be required for operation with a different collection.

We also notice that the correlation measure adopted with local context analysis is of type association. However, we already know that a correlation of type metric is expected to be more effective. Thus, it remains to be tested whether the adoption of a metric correlation factor (for the function $f(c, k_i)$) makes any difference with local context analysis.

## 5.4 Automatic Global Analysis

The methods of local analysis discussed above extract information from the local set of documents retrieved to expand the query. It is well accepted that such a procedure yields improved retrieval performance with various collections. An alternative approach is to expand the query using information from the whole set of documents in the collection. Strategies based on this idea are called global analysis procedures. Until the beginning of the 1990s, global analysis was considered to be a technique which failed to yield consistent improvements in retrieval performance with general collections. This perception has changed with the appearance of modern procedures for global analysis. In the following, we discuss two of these modern variants. Both of them are based on a thesaurus-like structure built using all the documents in the collection. However, the approach taken for building the thesaurus and the procedure for selecting terms for query expansion are quite distinct in the two cases.

### 5.4.1 Query Expansion based on a Similarity Thesaurus

In this section we discuss a query expansion model based on a global similarity thesaurus which is constructed automatically [655]. The similarity thesaurus is based on term to term relationships rather than on a matrix of co-occurrence (as discussed in section 5.3). The distinction is made clear in the discussion below. Furthermore, special attention is paid to the selection of terms for expansion and to the reweighting of these terms. In contrast to previous global analysis approaches, terms for expansion are selected based on their similarity to the whole query rather than on their similarities to individual query terms.

A *similarity thesaurus* is built considering term to term relationships. However, such relationships are not derived directly from co-occurrence of terms inside documents. Rather, they are obtained by considering that the terms are concepts in a concept space. In this concept space, each term is indexed by the documents in which it appears. Thus, terms assume the original role of documents while documents are interpreted as indexing elements. The following definitions establish the proper framework.

**Definition** *As before (see Chapter 2), let $t$ be the number of terms in the collection, $N$ be the number of documents in the collection, and $f_{i,j}$ be the frequency*

*of occurrence of the term $k_i$ in the document $d_j$. Further, let $t_j$ be the number of distinct index terms in the document $d_j$ and $itf_j$ be the inverse term frequency for document $d_j$. Then,*

$$itf_j = \log \frac{t}{t_j}$$

*analogously to the definition of inverse document frequency.*

Within this framework, to each term $k_i$ is associated a vector $\vec{k}_i$ given by

$$\vec{k}_i = (w_{i,1}, w_{i,2}, \ldots, w_{i,N})$$

where, as in Chapter 2, $w_{i,j}$ is a weight associated to the index-document pair $[k_i, d_j]$. Here, however, these weights are computed in a rather distinct form as follows.

$$w_{i,j} = \frac{(0.5 + 0.5\frac{f_{i,j}}{max_j(f_{i,j})})\ \ itf_j}{\sqrt{\sum_{l=1}^{N}(0.5 + 0.5\frac{f_{i,l}}{max_l(f_{i,l})})^2\ itf_j^2}} \tag{5.11}$$

where $max_j(f_{i,j})$ computes the maximum of all factors $f_{i,j}$ for the $i$-th term (i.e., over all documents in the collection). We notice that the expression above is a variant of tf-idf weights but one which considers inverse term frequencies instead.

The relationship between two terms $k_u$ and $k_v$ is computed as a correlation factor $c_{u,v}$ given by

$$c_{u,v} = \vec{k}_u \cdot \vec{k}_v = \sum_{\forall\ d_j} w_{u,j} \times w_{v,j} \tag{5.12}$$

We notice that this is a variation of the correlation measure used for computing scalar association matrices (defined by Equation 5.5). The main difference is that the weights are based on interpreting documents as indexing elements instead of repositories for term co-occurrence. The global similarity thesaurus is built through the computation of the correlation factor $c_{u,v}$ for each pair of indexing terms $[k_u, k_v]$ in the collection (analogously to the procedure in section 5.3). Of course, this is computationally expensive. However, this global similarity thesaurus has to be computed only once and can be updated incrementally.

Given the global similarity thesaurus, query expansion is done in three steps as follows.

- First, represent the query in the concept space used for representation of the index terms.

- Second, based on the global similarity thesaurus, compute a similarity $sim(q, k_v)$ between each term $k_v$ correlated to the query terms and the whole query $q$.
- Third, expand the query with the top $r$ ranked terms according to $sim(q, k_v)$.

For the first step, the query is represented in the concept space of index term vectors as follows.

**Definition** *To the query $q$ is associated a vector $\vec{q}$ in the term-concept space given by*

$$\vec{q} = \sum_{k_i \in q} w_{i,q} \vec{k}_i$$

*where $w_{i,q}$ is a weight associated to the index-query pair $[k_i, q]$. This weight is computed analogously to the index-document weight formula in equation 5.11.*

For the second step, a similarity $sim(q, k_v)$ between each term $k_v$ (correlated to the query terms) and the user query $q$ is computed as

$$sim(q, k_v) = \vec{q} \cdot \vec{k}_v = \sum_{k_u \in Q} w_{u,q} \times c_{u,v}$$

where $c_{u,v}$ is the correlation factor given in equation 5.12. As illustrated in Figure 5.2, a term might be quite close to the whole query while its distances to individual query terms are larger. This implies that the terms selected here for query expansion might be distinct from those selected by previous global analysis methods (which adopted a similarity to individual query terms for deciding terms for query expansion).

For the third step, the top $r$ ranked terms according to $sim(q, k_v)$ are added to the original query $q$ to form the expanded query $q'$. To each expansion term $k_v$ in the query $q'$ is assigned a weight $w_{v,q'}$ given by

$$w_{v,q'} = \frac{sim(q, k_v)}{\sum_{k_u \in q} w_{u,q}}$$

The expanded query $q'$ is then used to retrieve new documents to the user. This completes the technique for query expansion based on a similarity thesaurus. Contrary to previous global analysis approaches, this technique has yielded improved retrieval performance (in the range of 20%) with three different collections.

It is worthwhile making one final observation. Consider a document $d_j$ which is represented in the term-concept space by $\vec{d}_j = \sum_{k_i \in d_j} w_{i,j} \vec{k}_i$. Further, assume that the original query $q$ is expanded to include all the $t$ index terms

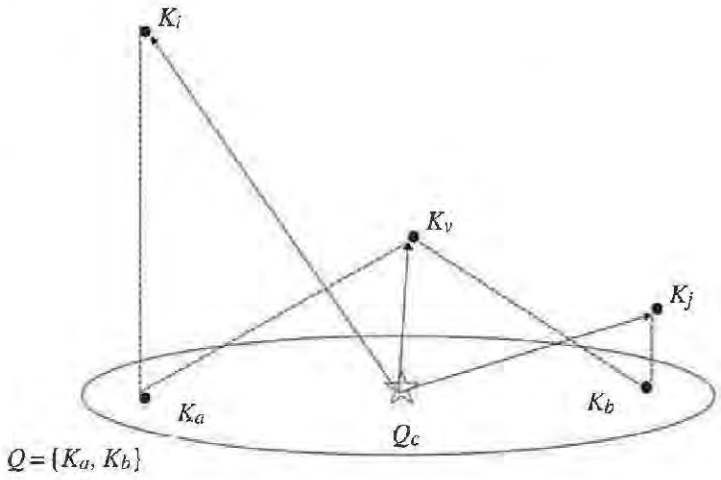



**Figure 5.2** The distance of a given term $k_v$ to the query centroid $Q_c$ might be quite distinct from the distances of $k_v$ to the individual query terms.

(properly weighted) in the collection. Then, the similarity $sim(q, d_j)$ between the document $d_j$ and the query $q$ can be computed in the term-concept space by

$$sim(q, d_j) \ \alpha \sum_{k_v \in d_j} \sum_{k_u \in q} w_{v,j} \times w_{u,q} \times c_{u,v}$$

Such an expression is analogous to the formula for query-document similarity in the generalized vector space model (see Chapter 2). Thus, the generalized vector space model can be interpreted as a query expansion technique. The main differences with the term-concept idea are the weight computation and the fact that only the top $r$ ranked terms are used for query expansion with the term-concept technique.

### 5.4.2 Query Expansion based on a Statistical Thesaurus

In this section, we discuss a query expansion technique based on a global statistical thesaurus [200]. Despite also being a global analysis technique, the approach is quite distinct from the one described above which is based on a similarity thesaurus.

The global thesaurus is composed of classes which group correlated terms in the context of the whole collection. Such correlated terms can then be used to expand the original user query. To be effective, the terms selected for expansion must have high term discrimination values [699] which implies that they must be low frequency terms. However, it is difficult to cluster low frequency terms effectively due to the small amount of information about them (they occur in few documents). To circumvent this problem, we cluster documents into

classes instead and use the low frequency terms in these documents to define our thesaurus classes. In this situation, the document clustering algorithm must produce small and tight clusters.

A document clustering algorithm which produces small and tight clusters is the *complete link algorithm* which works as follows (naive formulation).

(1) Initially, place each document in a distinct cluster.
(2) Compute the similarity between all pairs of clusters.
(3) Determine the pair of clusters $[C_u, C_v]$ with the highest inter-cluster similarity.
(4) Merge the clusters $C_u$ and $C_v$.
(5) Verify a stop criterion. If this criterion is not met then go back to step 2.
(6) Return a hierarchy of clusters.

The similarity between two clusters is defined as the minimum of the similarities between all pairs of inter-cluster documents (i.e., two documents not in the same cluster). To compute the similarity between documents in a pair, the cosine formula of the vector model is used. As a result of this minimality criterion, the resultant clusters tend to be small and tight.

Consider that the whole document collection has been clustered using the complete link algorithm. Figure 5.3 illustrates a small portion of the whole cluster hierarchy in which $sim(C_u, C_v) = 0.15$ and $sim(C_{u+v}, C_z) = 0.11$ where $C_{u+v}$ is a reference to the cluster which results from merging $C_u$ and $C_v$. Notice that the similarities decrease as we move up in the hierarchy because high level clusters include more documents and thus represent a looser grouping. Thus, the tightest clusters lie at the bottom of the clustering hierarchy.

Given the document cluster hierarchy for the whole collection, the terms that compose each class of the global thesaurus are selected as follows.

- Obtain from the user three parameters: threshold class (TC), number of

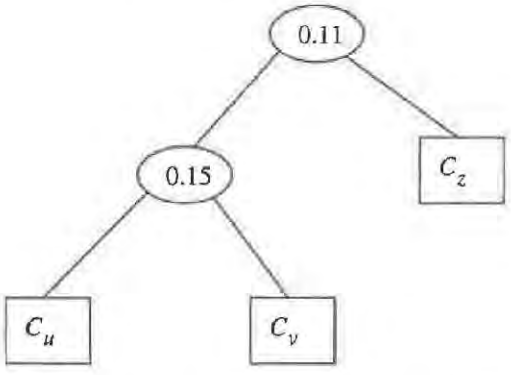



**Figure 5.3** Hierarchy of three clusters (inter-cluster similarities indicated in the ovals) generated by the complete link algorithm.

documents in a class (NDC), and minimum inverse document frequency (MIDF).

- Use the parameter TC as a threshold value for determining the document clusters that will be used to generate thesaurus classes. This threshold has to be surpassed by $sim(C_u, C_v)$ if the documents in the clusters $C_u$ and $C_v$ are to be selected as sources of terms for a thesaurus class. For instance, in Figure 5.3, a value of 0.14 for TC returns the thesaurus class $C_{u+v}$ while a value of 0.10 for TC returns the classes $C_{u+v}$ and $C_{u+v+z}$.
- Use the parameter NDC as a limit on the size of clusters (number of documents) to be considered. For instance, if both $C_{u+v}$ and $C_{u+v+z}$ are preselected (through the parameter TC) then the parameter NDC might be used to decide between the two. A low value of NDC might restrict the selection to the smaller cluster $C_{u+v}$.
- Consider the set of documents in each document cluster preselected above (through the parameters TC and NDC). Only the lower frequency documents are used as sources of terms for the thesaurus classes. The parameter MIDF defines the minimum value of inverse document frequency for any term which is selected to participate in a thesaurus class. By doing so, it is possible to ensure that only *low frequency* terms participate in the thesaurus classes generated (terms too generic are not good synonyms).

Given that the thesaurus classes have been built, they can be used for query expansion. For this, an average term weight $wt_C$ for each thesaurus class $C$ is computed as follows.

$$wt_C = \frac{\sum_{i=1}^{|C|} w_{i,C}}{|C|}$$

where $|C|$ is the number of terms in the thesaurus class $C$ and $w_{i,C}$ is a precomputed weight associated with the term-class pair $[k_i, C]$. This average term weight can then be used to compute a thesaurus class weight $w_C$ as

$$w_C = \frac{wt_C}{|C|} \times 0.5$$

The above weight formulations have been verified through experimentation and have yielded good results.

Experiments with four test collections (ADI, Medlars, CACM, and ISI; see Chapter 3 for details on these collections) indicate that global analysis using a thesaurus built by the complete link algorithm might yield consistent improvements in retrieval performance.

The main problem with this approach is the initialization of the parameters TC, NDC, and MIDF. The threshold value TC depends on the collection and can be difficult to set properly. Inspection of the cluster hierarchy is almost always necessary for assisting with the setting of TC. Care must be exercised because a

high value of TC might yield classes with too few terms while a low value of TC might yield too few classes. The selection of the parameter NDC can be decided more easily once TC has been set. However, the setting of the parameter MIDF might be difficult and also requires careful consideration.

## 5.5 Trends and Research Issues

The relevance feedback strategies discussed here can be directly applied to the graphical interfaces of modern information systems. However, since interactivity is now of greater importance, new techniques for capturing feedback information from the user are desirable. For instance, there is great interest in graphical interfaces which display the documents in the answer set as points in a 2D or 3D space. The motivation is to allow the user to quickly identify (by visual inspection) relationships among the documents in the answer. In this scenario, a rather distinct strategy for quantifying feedback information might be required. Thus, relevance strategies for dealing with visual displays are an important research problem.

In the past, global analysis was viewed as an approach which did not yield good improvements in retrieval performance. However, new results obtained at the beginning of the 1990s changed this perception. Further, the Web has provided evidence that techniques based on global analysis might be of interest to the users. For instance, this is the case with the highly popular 'Yahoo!' software which uses a manually built hierarchy of concepts to assist the user with forming the query. This suggests that investigating the utilization of global analysis techniques in the Web is a promising research problem.

Local analysis techniques are interesting because they take advantage of the local context provided with the query. In this regard, they seem more appropriate than global analysis techniques. Furthermore, many positive results have been reported in the literature. The application of local analysis techniques to the Web, however, has not been explored and is a promising research direction. The main challenge is the computational burden imposed on the search engine site due to the need to process document texts at query time. Thus, a related research problem of relevance is the development of techniques for speeding up query processing at the search engine site. In truth, this problem is of interest even if one considers only the normal processing of the queries because the search engines depend on processing as many queries as possible for economic survival.

The combination of local analysis, global analysis, visual displays, and interactive interfaces is also a current and important research problem. Allowing the user to visually explore the document space and providing him with clues which assist with the query formulation process are highly relevant issues. Positive results in this area might become a turning point regarding the design of user interfaces and are likely to attract wide attention.

## 5.6 Bibliographic Discussion

Query expansion methods have been studied for a long time. While the success of expansion methods throughout the years has been debatable, more recently researchers have reached the consensus that query expansion is a useful and little explored (commercially) technique. Useful because its modern variants can be used to consistently improve the retrieval performance with general collections. Little explored because few commercial systems (and Web search engines) take advantage of it.

Early work suggesting the expansion of a user query with closely related terms was done by Maron and Kuhns in 1960 [547]. The classic technique for combining query expansion with term reweighting in the vector model was studied by Rocchio in 1965 (using the Smart system [695] as a testbed) and published later on [678]. Ide continued the studies of Rocchio and proposed variations to the term reweighting formula [391].

The probabilistic model was introduced by Robertson and Sparck Jones [677] in 1976. A thorough and entertaining discussion of this model can be found in the book by van Rijsbergen [785]. Croft and Harper [199] suggested that the initial search should use a distinct computation. In 1983, Croft [198] proposed to extend the probabilistic formula to include within-document frequencies and introduced the $C$ and $K$ parameters.

Since the probabilistic model does not provide means of expanding the query, query expansion has to be done separately. In 1978, Harper and van Rijsbergen [345] used a term-term clustering technique based on a maximum spanning tree to select terms for probabilistic query expansion. Two years later, they also introduced a new relevance weighting scheme, called EMIM [344], to be used with their query expansion technique. In 1981, Wu and Salton [835] used relevance feedback to reweight terms (using a probabilistic formula) extracted from relevant documents and used these terms to expand the query. Empirical results showed improvements in retrieval performance.

Our discussion on user relevance feedback for the vector and probabilistic models in section 5.2 is based on four sources: a nice paper by Salton and Buckley [696], the book by van Rijsbergen [785], the book by Salton and McGill [698], and two book chapters by Harman [340, 339].

Regarding automatic query expansion, Lesk [500] tried variations of term-term clustering in the Smart system without positive results. Following that, Sparck Jones and Barber [413] and Minker, Wilson and Zimmerman [562] also observed no improvements with query expansion based on term-term global clustering. These early research results left the impression that query expansion based on global analysis is not an effective technique. However, more recent results show that this is not the case. In fact, the research results obtained by Vorhees [793], by Crouch and Yang [200], and by Qiu and Frei [655] indicate that query expansion based on global analysis can consistently yield improved retrieval performance.

Our discussion on query expansion through local clustering is based on early work by Attar and Fraenkel [35] from 1977. The idea of local context

analysis is much more recent and was introduced by Xu and Croft [838] in 1996. The discussion on query expansion using a global similarity thesaurus is based on the work by Qiu and Frei [655]. Finally, the discussion on query expansion using a global statistical thesaurus is based on the work of Crouch and Yang [200] which is influenced by the term discrimination value theory introduced by Salton, Yang, and Yu [699] early in 1975.

Since query expansion frequently is based on some form of clustering, our discussion covered a few clustering algorithms. However, our aim was not to provide a thorough review of clustering algorithms for information retrieval. Such a review can be found in the work of Rasmussen [668].

# Chapter 6
# Text and Multimedia Languages and Properties

## 6.1 Introduction

Text is the main form of communicating knowledge. Starting with hieroglyphs, the first written surfaces (stone, wood, animal skin, papyrus, and rice paper), and paper, text has been created everywhere, in many forms and languages. We use the term *document* to denote a single unit of information, typically text in a digital form, but it can also include other media. In practice, a document is loosely defined. It can be a complete logical unit, like a research article, a book or a manual. It can also be part of a larger text, such as a paragraph or a sequence of paragraphs (also called a *passage* of text), an entry in a dictionary, a judge's opinion on a case, the description of an automobile part, etc. Furthermore, with respect to its physical representation, a document can be any physical unit, for example a file, an email, or a World Wide Web (or just Web) page.

A document has a given syntax and structure which is usually dictated by the application or by the person who created it. It also has a semantics, specified by the author of the document (who is not necessarily the same as the creator). Additionally, a document may have a presentation style associated with it, which specifies how it should be displayed or printed. Such a style is usually given by the document syntax and structure and is related to a specific application (for example, a Web browser). Figure 6.1 depicts all these relations. A document can also have information about itself, called *metadata*. The next section explains different types of metadata and their relevance.

The *syntax* of a document can express structure, presentation style, semantics, or even external actions. In many cases one or more of these elements are implicit or are given together. For example, a structural element (e.g., a section) can have a fixed formatting style. The semantics of a document is also associated with its use. For example, Postscript directives are designed for drawing.

The syntax of a document can be implicit in its content, or expressed in a simple declarative language or even in a programming language. For example, many editor formats are declarative while a TeX document uses a powerful typesetting language. Although a powerful language could be easier to parse than the data itself, it might be difficult to convert documents in that language to other formats. Many syntax languages are proprietary and specific, but open

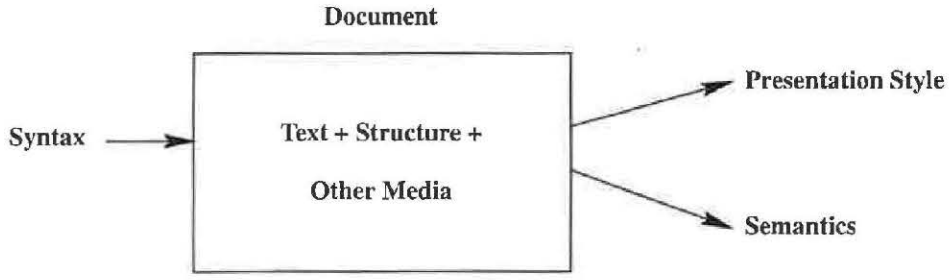



**Figure 6.1** Characteristics of a document.

and generic languages are better because documents can be interchanged between applications and are more flexible. Text can also be written in natural language. However, at present the semantics of natural language is still not easy for a computer to understand. The current trend is to use languages which provide information on the document structure, format, and semantics while being readable by humans as well as computers. The Standard Generalized Markup Language (SGML), which is covered later on in this chapter, tries to balance all the issues above. Metadata, markup, and semantic encoding represent different levels of formalization of the document contents.

Most documents have a particular formatting style. However, new applications are pushing for external formatting such that information can be represented independently of style, and vice versa. The presentation style can be embedded in the document, as in TeX or Rich Text Format (RTF). Style can be complemented by macros (for example, LaTeX in the case of TeX). In most cases, style is defined by the document author. However, the reader may decide part of the style (for example, by setting options in a Web browser). The style of a document defines how the document is visualized in a computer window or a printed page, but can also include treatment of other media such as audio or video.

In this chapter we first cover metadata. Following that we discuss text characteristics such as formats and natural language statistics. Next we cover languages to describe text structure, presentation style, or semantics. The last part is devoted to multimedia formats and languages.

## 6.2 Metadata

Most documents and text collections have associated with them what is known as metadata. Metadata is information on the organization of the data, the various data domains, and the relationship between them. In short, metadata is 'data about the data.' For instance, in a database management system, the schema

specifies some of the metadata, namely, the name of the relations, the fields or attributes of each relation, the domain of each attribute, etc.

Common forms of metadata associated with text include the author, the date of publication, the source of the publication, the document length (in pages, words, bytes, etc.), and the document genre (book, article, memo, etc.). For example, the Dublin Core Metadata Element Set [807] proposes 15 fields to describe a document. Following Marchionini [542], we refer to this kind of information as Descriptive Metadata, metadata that is external to the meaning of the document, and pertains more to how it was created. Another type of metadata characterizes the subject matter that can be found within the document's contents. We will refer to this as Semantic Metadata. Semantic Metadata is associated with a wide number of documents and its availability is increasing. All books published within the USA are assigned Library of Congress subject codes, and many journals require author-assigned key terms that are selected from a closed vocabulary of relevant terms. For example, biomedical articles that appear within the MEDLINE (see Chapter 3) system are assigned topical metadata pertaining to disease, anatomy, pharmaceuticals, and so on. To standardize semantic terms, many areas use specific *ontologies*, which are hierarchical taxonomies of terms describing certain knowledge topics.

An important metadata format is the Machine Readable Cataloging Record (MARC) which is the most used format for library records. MARC has several fields for the different attributes of a bibliographic entry such as title, author, etc. Specific uses of MARC are given in Chapter 14. In the USA, a particular version of MARC is used: USMARC, which is an implementation of ANSI/NISO Z39.2, the American National Standard for Bibliographic Information Interchange. The USMARC format documents contain the definitions and content for the fields that have to be used in records structured according to Z39.2. This standard is maintained by the Library of Congress of the USA.

With the increase of data in the Web, there are many initiatives to add metadata information to Web documents. In the Web, metadata can be used for many purposes. Some of them are cataloging (BibTeX is a popular format for this case), content rating (for example, to protect children from reading some type of documents), intellectual property rights, digital signatures (for authentication), privacy levels (who should and who should not have access to a document), applications to electronic commerce, etc. The new standard for Web metadata is the Resource Description Framework (RDF), which provides interoperability between applications. This framework allows the description of Web resources to facilitate automated processing of the information. It does not assume any particular application or semantic domain. It consists of a description of nodes and attached attribute/value pairs. Nodes can be any Web resource, that is, any Uniform Resource Identifier (URI), which includes the Uniform Resource Locator (URL). Attributes are properties of nodes, and their values are text strings or other nodes (Web resources or metadata instances). To describe the semantics, values from, for example, the Dublin Core library metadata URL can be used. Other predefined vocabularies for authoring metadata are expected, in particular for content rating and for digital signatures. In addition, currently,

there are many Web projects on ontologies for different application domains (see also Chapters 13 and 15). Metadata is also useful for metadescriptions of non-textual objects. For example, a set of keywords that describe an image. These keywords can later be used to search for the image using classic text information retrieval techniques (on the metadescriptions).

## 6.3 Text

With the advent of the computer, it was necessary to code text in binary digits. The first coding schemes were EBCDIC and ASCII, which used seven bits to code each possible symbol. Later, ASCII was standardized to eight bits (ISO-Latin), to accommodate several languages, including accents and other diacritical marks. Nevertheless, ASCII is not suitable for oriental languages such as Chinese or Japanese Kanji, where each symbol might represent a concept and therefore thousands of them exist. For this case, a 16-bit code exists called Unicode (ISO 10616) [783].

In this section we cover different characteristics of text. First, the possible formats of text, ASCII being the simplest format. Second, how the information content of text can be measured, followed by different models for it. Finally, we mention briefly how we can measure similarity between strings or pieces of text.

### 6.3.1 Formats

There is no single format for a text document, and an IR system should be able to retrieve information from many of them. In the past, IR systems would convert a document to an internal format. However, that has many disadvantages, because the original application related to the document is not useful any more. On top of that, we cannot change the contents of a document. Current IR systems have filters that can handle most popular documents, in particular those of word processors with some binary syntax such as Word, WordPerfect or FrameMaker. Even then, good filters might not be possible if the format is proprietary and its details are not public. This is not the case for full ASCII syntax, as in TeX documents. Although documents can be in a binary format (for example, parts of a Word document), documents that are represented in human-readable ASCII form imply more portability and are easier to modify (for example, they can be edited with different applications).

Other text formats were developed for document interchange. Among these we should mention the Rich Text Format (RTF), which is used by word processors and has ASCII syntax. Other important formats were developed for displaying or printing documents. The most popular ones are the Portable Document Format (PDF) and Postscript (which is a powerful programming language for drawing). Other interchange formats are used to encode electronic mail, for example MIME (Multipurpose Internet Mail Exchange). MIME supports multiple character sets, multiple languages, and multiple media.

On top of these formats, nowadays many files are compressed. Text compression is treated in detail in Chapter 7, but here we comment on the most

popular compression software and associated formats. These include Compress (Unix), ARJ (PCs), and ZIP (for example `gzip` in Unix and `Winzip` in Windows). Other tools allow us to convert binary files, in particular compressed text, to ASCII text such that it can be transmitted through a communication line using only seven bits. Examples of these tools are `uuencode/uudecode` and `binhex`.

### 6.3.2 Information Theory

Written text has a certain semantics and is a way to communicate information. Although it is difficult to formally capture how much information is there in a given text, the distribution of symbols is related to it. For example, a text where one symbol appears almost all the time does not convey much information. Information theory defines a special concept, *entropy*, to capture information content (or equivalently, information uncertainty). If the alphabet has $\sigma$ symbols, each one appearing with probability $p_i$ (probability here is defined as the symbol frequency over the total number of symbols) in a text, the entropy of this text is defined as

$$E = -\sum_{i=1}^{\sigma} p_i \log_2 p_i$$

In this formula the $\sigma$ symbols of the alphabet are coded in binary, so the entropy is measured in bits. As an example, for $\sigma = 2$, the entropy is 1 if both symbols appear the same number of times or 0 if only one symbol appears. We say that the amount of information in a text can be quantified by its entropy. The definition of entropy depends on the probabilities (frequencies) of each symbol. To obtain those probabilities we need a *text model*. So we say that the amount of information in a text is measured with regard to the text model. This concept is also important, for example, in text compression, where the entropy is a limit on how much the text can be compressed, depending on the text model.

In our case we are interested in natural language, as we now discuss.

### 6.3.3 Modeling Natural Language

Text is composed of symbols from a finite alphabet. We can divide the symbols in two disjoint subsets: symbols that separate words and symbols that belong to words. It is well known that symbols are not uniformly distributed. If we consider just letters (a to z), we observe that vowels are usually more frequent than most consonants. For example, in English, the letter 'e' has the highest frequency. A simple model to generate text is the binomial model. In it, each symbol is generated with a certain probability. However, natural language has a dependency on previous symbols. For example, in English, a letter 'f' cannot appear after a letter 'c' and vowels or certain consonants have a higher probability

of occurring. Therefore, the probability of a symbol depends on previous symbols. We can use a finite-context or Markovian model to reflect this dependency. The model can consider one, two, or more letters to generate the next symbol. If we use $k$ letters, we say that it is a $k$-order model (so the binomial model is considered a 0-order model). We can use these models taking words as symbols. For example, text generated by a 5-order model using the distribution of words in the Bible might make sense (that is, it can be grammatically correct), but will be different from the original. More complex models include finite-state models (which define regular languages), and grammar models (which define context free and other languages). However, finding the right grammar for natural language is still a difficult open problem.

The next issue is how the different words are distributed inside each document. An approximate model is *Zipf's Law* [847, 310], which attempts to capture the distribution of the frequencies (that is, number of occurrences) of the words in the text. The rule states that the frequency of the $i$-th most frequent word is $1/i^\theta$ times that of the most frequent word. This implies that in a text of $n$ words with a vocabulary of $V$ words, the $i$-th most frequent word appears $n/(i^\theta H_V(\theta))$ times, where $H_V(\theta)$ is the harmonic number of order $\theta$ of $V$, defined as

$$H_V(\theta) = \sum_{j=1}^{V} \frac{1}{j^\theta}$$

so that the sum of all frequencies is $n$. The left side of Figure 6.2 illustrates the distribution of frequencies considering that the words are arranged in decreasing order of their frequencies. The value of $\theta$ depends on the text. In the most simple formulation, $\theta = 1$, and therefore $H_V(\theta) = O(\log n)$. However, this simplified version is very inexact, and the case $\theta > 1$ (more precisely, between 1.5 and 2.0) fits better the real data [26]. This case is very different, since the distribution is much more skewed, and $H_V(\theta) = O(1)$. Experimental data suggests that a better model is $k/(c+i)^\theta$ where $c$ is an additional parameter and $k$ is such that all frequencies add to $n$. This is called a Mandelbrot distribution [561].

Since the distribution of words is very skewed (that is, there are a few hundred words which take up 50% of the text), words that are too frequent, such as *stopwords*, can be disregarded. A stopword is a word which does not carry meaning in natural language and therefore can be ignored (that is, made not searchable), such as 'a,' 'the,' 'by,' etc. Fortunately the most frequent words are stopwords and therefore, half of the words appearing in a text do not need to be considered. This allows us, for instance, to significantly reduce the space overhead of indices for natural language texts. For example, the most frequent words in the TREC-2 collection (see Chapter 3 for details on this reference collection and others) are 'the,' 'of,' 'and,' 'a,' 'to' and 'in' (see also Chapter 7).

Another issue is the distribution of words in the documents of a collection. A simple model is to consider that each word appears the same number of times in every document. However, this is not true in practice. A better model is

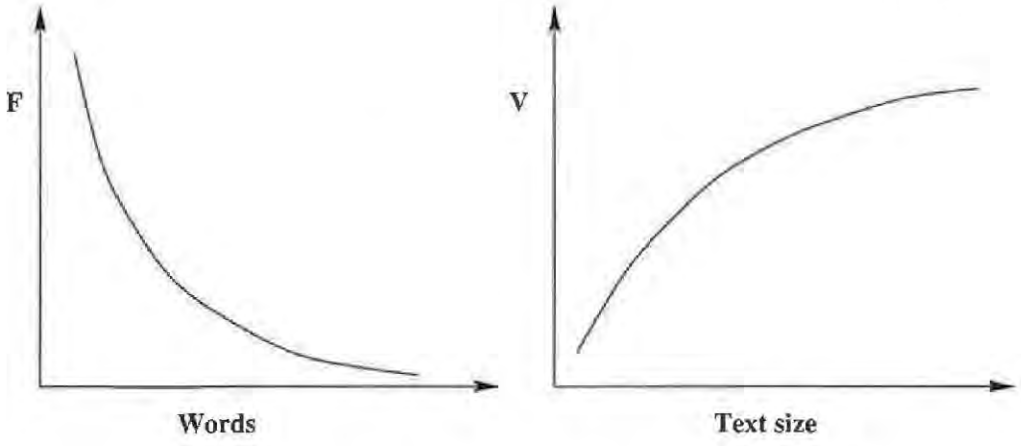



**Figure 6.2** Distribution of sorted word frequencies (left) and size of the vocabulary (right).

to consider a negative binomial distribution, which says that the fraction of documents containing a word $k$ times is

$$F(k) = \binom{\alpha + k - 1}{k} p^k (1+p)^{-\alpha - k}$$

where $p$ and $\alpha$ are parameters that depend on the word and the document collection. For example, for the Brown Corpus [276] and the word 'said', we have $p = 9.24$ and $\alpha = 0.42$ [171]. The latter reference gives other models derived from a Poisson distribution.

The next issue is the number of distinct words in a document. This set of words is referred to as the document *vocabulary*. To predict the growth of the vocabulary size in natural language text, we use the so-called *Heaps' Law* [352]. This is a very precise law which states that the vocabulary of a text of size $n$ words is of size $V = Kn^\beta = O(n^\beta)$, where $K$ and $\beta$ depend on the particular text. The right side of Figure 6.2 illustrates how the vocabulary size varies with the text size. $K$ is normally between 10 and 100, and $\beta$ is a positive value less than one. Some experiments [26, 42] on the TREC-2 collection show that the most common values for $\beta$ are between 0.4 and 0.6. Hence, the vocabulary of a text grows sublinearly with the text size, in a proportion close to its square root.

Notice that the set of different words of a language is fixed by a constant (for example, the number of different English words is finite). However, the limit is so high that it is much more accurate to assume that the size of the vocabulary is $O(n^\beta)$ instead of $O(1)$, although the number should stabilize for huge enough texts. On the other hand, many authors argue that the number keeps growing anyway because of typing or spelling errors.

Heaps' law also applies to collections of documents because, as the total text size grows, the predictions of the model become more accurate. Furthermore, this model is also valid for the World Wide Web (see Chapter 13).

The last issue is the average length of words. This relates the text size in

words with the text size in bytes (without accounting for punctuation and other extra symbols). For example, in the different subcollections of the TREC-2 collection, the average word length is very close to 5 letters, and the range of variation of this average in each subcollection is small (from 4.8 to 5.3 letters). If we remove the stopwords, the average length of a word increases to a number between 6 and 7 (letters). If we take only the words of the vocabulary, the average length is higher (about 8 or 9). This defines the total space needed for the vocabulary.

Heaps' law implies that the length of the words in the vocabulary increases logarithmically with the text size and thus, that longer and longer words should appear as the text grows. However, in practice, the average length of the words in the overall text is constant because shorter words are common enough (e.g. stopwords). This balance between short and long words, such that the average word length remains constant, has been noticed many times in different contexts, and can also be explained by a finite-state model in which: (a) the space character has probability close to 0.2; (b) the space character cannot appear twice subsequently; and (c) there are 26 letters [561]. This simple model is consistent with Zipf's and Heaps' laws.

The models presented in this section are used in Chapters 8 and 13, in particular Zipf's and Heaps' laws.

### 6.3.4 Similarity Models

In this section we define notions of syntactic similarity between strings or documents. Similarity is measured by a *distance function*. For example, if we have strings of the same length, we can define the distance between them as the number of positions that have different characters. Then, the distance is 0 if they are equal. This is called the Hamming distance. A distance function should also be symmetric (that is, the order of the arguments does not matter) and should satisfy the triangle inequality (that is, $distance(a, c) \leq distance(a, b) + distance(b, c)$).

An important distance over strings is the edit or Levenshtein distance mentioned earlier. The edit distance is defined as the minimum number of characters, insertions, deletions, and substitutions that we need to perform in any of the strings to make them equal. For instance, the edit distance between 'color' and 'colour' is one, while the edit distance between 'survey' and 'surgery' is two. The edit distance is considered to be superior for modeling syntactic errors than other more complex methods such as the Soundex system, which is based on phonetics [595]. Extensions to the concept of edit distance include different weights for each operation, adding transpositions, etc.

There are other measures. For example, assume that we are comparing two given strings and the only operation allowed is deletion of characters. Then, after all non-common characters have been deleted, the remaining sequence of characters (not necessarily contiguous in the original string, but in the same order) is the longest common subsequence (LCS) of both strings. For example, the LCS of 'survey' and 'surgery' is 'surey.'

Similarity can be extended to documents. For example, we can consider lines as single symbols and compute the longest common sequence of lines between two files. This is the measure used by the `diff` command in Unix-like operating systems. The main problem with this approach is that it is very time consuming and does not consider lines that are similar. The latter drawback can be fixed by taking a weighted edit distance between lines or by computing the LCS over all the characters. Other solutions include extracting fingerprints (any piece of text that in some sense characterizes it) for the documents and comparing them, or finding large repeated pieces. There are also visual tools to see document similarity. For example, `Dotplot` draws a rectangular map where both coordinates are file lines and the entry for each coordinate is a gray pixel that depends on the edit distance between the associated lines.

## 6.4 Markup Languages

Markup is defined as extra textual syntax that can be used to describe formatting actions, structure information, text semantics, attributes, etc. For example, the formatting commands of TeX (a popular text formatting software) could be considered markup. However, formal markup languages are much more structured. The marks are called tags, and usually, to avoid ambiguity, there is an initial and ending tag surrounding the marked text. The standard metalanguage for markup is SGML, as already mentioned. An important subset of SGML is XML (eXtensible Markup Language), the new metalanguage for the Web. The most popular markup language used for the Web, HTML (HyperText Markup Language), is an instance of SGML. All these languages and examples of them are described below.

### 6.4.1 SGML

SGML stands for Standard Generalized Markup Language (ISO 8879) and is a metalanguage for tagging text developed by a group led by Goldfarb [303] based on earlier work done at IBM. That is, SGML provides the rules for defining a markup language based on tags. Each instance of SGML includes a description of the document structure called a document type definition. Hence, an SGML document is defined by: (1) a description of the structure of the document and (2) the text itself marked with tags which describe the structure. We will explain later the syntax associated with the tags.

The document type definition is used to describe and name the pieces that a document is composed of and define how those pieces relate to each other. Part of the definition can be specified by an SGML document type declaration (DTD).

Other parts, such as the semantics of elements and attributes, or application conventions, cannot be expressed formally in SGML. Comments can be used, however, to express them informally. This means that all of the rules for applying SGML markup to documents are part of the definition, and those that can be expressed in SGML syntax are represented in the DTD. The DTD does not define the semantics (that is, the meaning, presentation, and behavior), or intended use, of the tags. However, some semantic information can be included in comments embedded in the DTD, while more complete information is usually present in separate documentation. This additional documentation typically describes the elements, or logical pieces of data, the attributes, and information about those pieces of data. For example, two tags can have the same name but different semantics in two different applications.

Tags are denoted by angle brackets (`<tagname>`). Tags are used to identify the beginning and ending of pieces of the document, for example a quote in a literary text. Ending tags are specified by adding a slash before the tag name (e.g., `</tagname>`). For example, the tag `</author>` could be used to identify the element 'name of author,' which appears in italics and generates a link to a biographic sketch. Tag attributes are specified at the beginning of the element, inside the angle brackets and after the nametag using the syntax `attname=value`.

Figure 6.3 gives an example of a simple DTD and a document using it. While we do not intend to discuss SGML syntax here, we give a brief description of the example such that the reader can grasp the main ideas. Each ELEMENT represents a tag denoted by its name. The two following characters indicate if the starting and ending tags are compulsory (-) or optional (O). For example, the ending tag for `prolog` is necessary while for `sender` it is not. Following that, the inside portion of the content tag is specified using a regular expression style syntax where ',' stands for concatenation, '|' stands for logical or, '?' stands for zero or one occurrence, '*' stands for zero or more occurrences, and '+' stands for one or more occurrences of the preceding element. The content tag can be composed of the combination of other tag contents, ASCII characters (PCDATA), and binary data (NDATA), or EMPTY. The possible attributes of a tag are given in an attribute list (ATTLIST) identified by the tag name, followed by the name of each attribute, its type, and if it is required or not (otherwise, the default value is given). An SGML document instance is associated with the DTD so that the various tools working with the data know which are the correct tags and how they are organized.

The document description generally does not specify how a document should look, for example when it is printed on paper or displayed on a screen. Because SGML separates content from format, we can create very good models of data that have no mechanism for describing the format, hence, no standard way to output the data in a formatted fashion. Therefore, output specifications, which are directions on how to format a document, are often added to SGML documents. For this purpose, output specification standards such as DSSSL (Document Style Semantic Specification Language) and FOSI (Formatted Output Specification Instance) were devised. Both of these standards define mechanisms for associating style information with SGML document instances.

```
<!--SGML DTD for electronic messages -->

<!ELEMENT e-mail          - - (prolog, contents) >
<!ELEMENT prolog          - - (sender, address+, subject?, Cc*) >
<!ELEMENT (sender | address | subject | Cc) - O (#PCDATA) >
<!ELEMENT contents        - - (par | image | audio)+ >
<!ELEMENT par             - O (ref | #PCDATA)+ >
<!ELEMENT ref             - O EMPTY >
<!ELEMENT (image | audio) - -  (#NDATA) >

<!ATTLIST e-mail
          id              ID                    #REQUIRED
          date_sent       DATE                  #REQUIRED
          status          (secret | public )    public >
<!ATTLIST ref
          id              IDREF                 #REQUIRED >
<!ATTLIST (image | audio )
          id              ID                    #REQUIRED >

<!--Example of use of previous DTD-->
<!DOCTYPE e-mail SYSTEM "e-mail.dtd">
<e-mail id=94108rby date_sent=02101998>
  <prolog>
  <sender> Pablo Neruda </sender>
  <address> Federico García Lorca </address>
  <address> Ernest Hemingway </address>
  <subject> Pictures of my house in Isla Negra
  <Cc> Gabriel García Márquez </Cc>
  </prolog>
  <contents>
      <par>
      As promised in my previous letter, I am sending two digital
      pictures to show you my house and the splendid view of the
      Pacific Ocean from my bedroom (photo <ref idref=F2>).
      </par>
      <image id=F1> "photo1.gif" </image>
      <image id=F2> "photo2.jpg" </image>
      <par>
      Regards from the South, Pablo.
  </contents>
</e-mail>
```

**Figure 6.3** DTD for structuring electronic mails and an example of its use.

They are the components of an SGML system used for defining, for instance, that the data identified by a tag should be typeset in italics.

One important use of SGML is in the Text Encoding Initiative (TEI). The TEI is a cooperative project that started in 1987 and includes several US associations related to the humanities and linguistics. The main goal is to generate guidelines for the preparation and interchange of electronic texts for scholarly

research, as well as for industry. In addition to the guidelines, TEI provides several document formats through SGML DTDs. One of the most used formats is TEI Lite. The TEI Lite DTD can be used stand-alone or together with the full set of TEI DTD files.

### 6.4.2 HTML

HTML stands for HyperText Markup Language and is an instance of SGML. HTML was created in 1992 and has evolved during the past years, 4.0 being the latest version, released as a recommendation at the end of 1997. Currently it is being extended in many ways to solve its many limitations, for example, to be able to write mathematical formulas. Most documents on the Web are stored and transmitted in HTML. HTML is a simple language well suited for hypertext, multimedia, and the display of small and simple documents.

HTML is based on SGML, and although there is an HTML DTD (Document Type Definition), most HTML instances do not explicitly make reference to the DTD. The HTML tags follow all the SGML conventions and also include formatting directives.

HTML documents can have other media embedded within them, such as images or audio in different formats. HTML also has fields for metadata, which can be used for different applications and purposes. If we also add programs (for example, using Javascript) inside a page, some people call it dynamic HTML (or DHTML). This should not be confused with a Microsoft proposal (also called dynamic HTML) of an Application Programming Interface (API) for accessing and manipulating HTML documents. Figure 6.4 gives an example of an HTML document together with its output in a Web browser.

Because HTML does not fix the presentation style of a document, in 1997, Cascade Style Sheets (CSS) were introduced. CSS offer a powerful and manageable way for authors, artists, and typographers to create visual effects that improve the aesthetics of HTML pages in the Web. Style sheets can be used one after another (called cascading) to define the presentation style for different elements of an HTML page. Style sheets separate information about presentation from document content, which in turn simplifies Web site maintenance, promotes Web page accessibility, and makes the Web faster. However, CSS support in current browsers is still modest. Another disadvantage is that two style sheets do not have to be consistent nor complete, so the stylistic result might not be good, in particular regarding color. CSS are supposed to balance the expectations of the author and of the reader regarding presentation issues. Nevertheless, it is not clear who or in which cases the author or the reader should define the presentation.

The evolution of HTML implies support for backward compatibility and also for forward compatibility, because people should also be able to see new documents with old browsers. HTML 4.0 has been specified in three flavors: strict, transitional, and frameset. Strict HTML only worries about non-presentational

```
<html>
<head>
<title>HTML Example</title>
<meta name=rby content="Just an example">
</head>
<body>
<h1>HTML Example</h1>
<p>
<hr>
<p>
HTML has many <i>tags</i>, among them:
<ul>
<li> links to other <a href=http://www.w3c.org/>pages</a>
    (a from anchor),
<li> paragraphs (p), headings (h1, h2, etc), font types (b, i),
<li> horizontal rules (hr), indented lists and items (ul, li),
<li> images (img), tables, forms, etc.
</ul>
<p>
<hr>
<p>
<img align=left src="at_work.gif">
This page is <b>always</b> under construction.
</body>
</html>
```

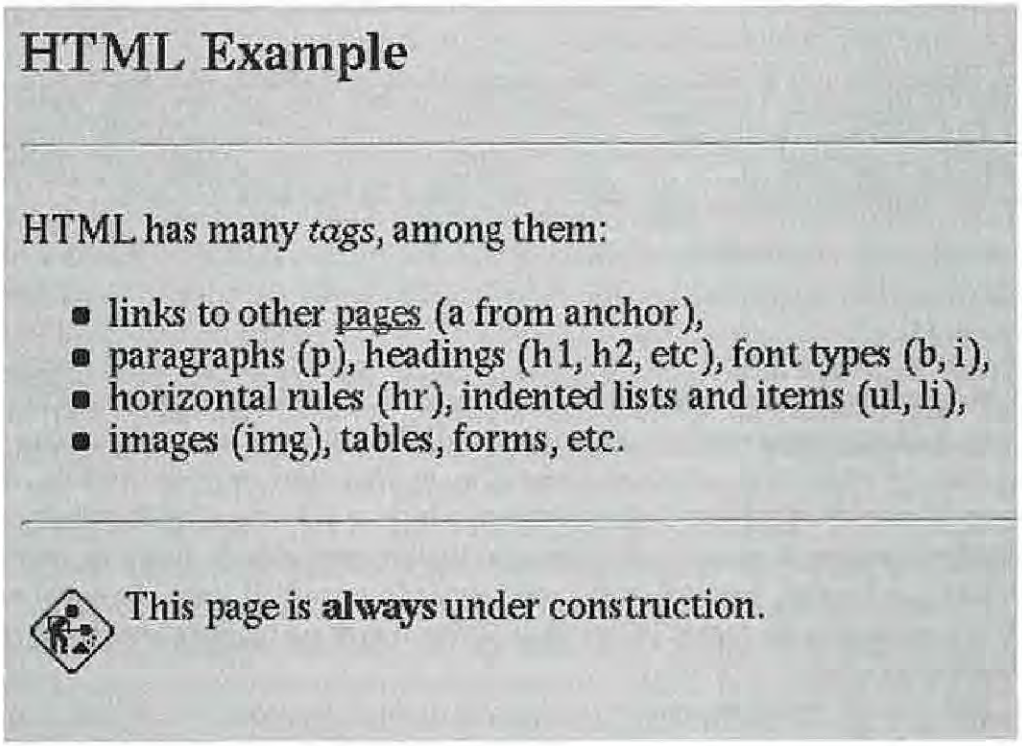



**Figure 6.4** Example of an HTML document and how it is seen in a browser.

markup, leaving all the displaying information to CSS. Transitional HTML uses all the presentational features for pages that should be read for old browsers that do not understand CSS. Frameset HTML is used when you want to partition the browser window in two or more frames. HTML 4.0 includes support for style sheets, internationalization, frames, richer tables and forms, and accessibility options for people with disabilities.

Typical HTML applications use a fixed small set of tags in conformance with a single SGML specification. Fixing a small set of tags allows users to leave the language specification out of the document and makes it much easier to build applications, but this advantage comes at the cost of severely limiting HTML in several important aspects. In particular, HTML does not:

- allow users to specify their own tags or attributes in order to parameterize or otherwise semantically qualify their data;
- support the specification of nested structures needed to represent database schemas or object-oriented hierarchies;
- support the kind of language specification that allows consuming applications to check data for structural validity on importation.

In contrast to HTML stands generic SGML. A generic SGML application is one that supports SGML language specifications of arbitrary complexity and makes possible the qualities of extensibility, structure, and validation missing in HTML. SGML makes it possible to define your own formats for your own documents, to handle large and complex documents, and to manage large information repositories. However, full SGML contains many optional features that are not needed for Web applications and have proven to have a cost/benefit ratio unattractive to current vendors of Web browsers. All these reasons led to the development of XML, a simpler metalanguage that is described in the next section.

### 6.4.3 XML

XML stands for eXtensible Markup Language and is a simplified subset of SGML. That is, XML is not a markup language, as HTML is, but a metalanguage that is capable of containing markup languages in the same way as SGML. XML allows a human-readable semantic markup, which is also machine-readable. As a result, XML makes it easier to develop and deploy new specific markup, enabling automatic authoring, parsing, and processing of networked data. In some ways, XML allows one to do many things that today are done by Java scripts or other program interfaces.

XML does not have many of the restrictions imposed by HTML but on the other hand imposes a more rigid syntax on the markup, which becomes important at processing time. In XML, ending tags cannot be omitted. Also, tags for elements that do not have any content, like BR and IMG, are specially marked by a slash before the closing angle bracket. XML also distinguishes upper

```
<?XML VERSION="1.0" RMD="NONE" ?>
<e-mail id="94108rby" date_sent="02101998">
  <prolog>
  <sender> Pablo Neruda </sender>
  <address> Federico García Lorca </address>
  <address> Ernest Hemingway </address>
  <subject> Pictures of my house in Isla Negra
  <Cc> Gabriel García Márquez </Cc>
  </prolog>
  <contents>
      <par>
      As promised in my previous letter, I am sending two digital
      pictures to show you my house and the splendid view of the
      Pacific Ocean from my bedroom (photo <ref idref="F2"/>).
      </par>
      <image id="F1" ref="photo1.gif" />
      <image id="F2"> ref="photo2.jpg" />
      <par>
      Regards from the South, Pablo.
      </par>
  </contents>
</e-mail>
```

**Figure 6.5** An XML document without a DTD analogous to the previous SGML example.

and lower case, so `img` and `IMG` are different tags (this is not true in HTML). In addition, all attribute values must be between quotes. This implies that parsing XML without knowledge of the tags is easier. In particular, using a DTD is optional. If there is no DTD, the tags are obtained while the parsing is done. With respect to SGML, there are a few syntactic differences, and many more restrictions. Listing all these differences is beyond the scope of this book, but Figure 6.5 shows an example of a DTDless XML document based on the previous electronic mail DTD given for SGML (see Figure 6.3). The `RMD` attribute stands for Required Markup Declaration, which indicates whether a DTD must be used or not (no DTD in this case). Other possible values are `INTERNAL` which means that the DTD is inside the document or `ALL` (default value) which allows the use of external sources for part or the whole DTD as in SGML.

XML allows any user to define new tags, define more complex structures (for example, unbounded nesting with the same rules of SGML) and has data validation capabilities. As XML is very new, there is still some discussion of how it will change or impact Internet applications. XML is a profile of SGML that eliminates many of the difficulties of implementing things, so for the most part it behaves just like SGML, as shown before. As mentioned, XML removes the requirement for the existence of a DTD, which can be parsed directly from the data. Removing the DTD places even more importance on the application documentation. This can also have a large impact on the functions that the software

provides. For example, it means that if an XML editor does not use a DTD, how will it help the user to tag the documents consistently? These problems should be resolved in the near future. In the case of semantic ambiguity between tag names, one goal is to have a *namespace* such that there is a convention for its use.

The Extensible Style sheet Language (XSL) is the XML counterpart of Cascading Style Sheets. XSL is designed to transform and style highly-structured, data-rich documents written in XML. For example, with XSL it would be possible to automatically extract a table of contents from a document. The syntax of XSL has been defined using XML. In addition to adding style to a document, XSL can be used to transform XML documents to HTML and CSS. This is analogous to macros in a word processor.

Another extension to XML, defined using XML, is the Extensible Linking Language (XLL). XLL defines different types of links, including external and internal links. In particular, any element type can be the origin of a link and outgoing links can be defined on documents that cannot be modified. The behavior of the links is also more generic. The object linked can be embedded in, or replace the document. It is also possible to generate a new context without changing the current application (for example, the object is displayed in a new window).

Recent uses of XML include:

- **Mathematical Markup Language** (MathML): two sets of tags, one for presentation of formulas and another for the meaning of mathematical expressions.
- **Synchronized Multimedia Integration Language** (SMIL): a declarative language for scheduling multimedia presentations in the Web, where the position and activation time of different objects can be specified.
- **Resource Description Format** (already covered in section 6.2): metadata information for XML should be given using RDF.

The XML movement is one indication that a parseable, hierarchical object model will play an increasingly major role in the evolution of HTML. The next generation of HTML should be based on a suite of XML tag sets to be used together with mathematics, synchronized multimedia, and vector graphics (possibly using the XML-based languages already mentioned). That is, the emphasis will be on structuring and modeling data rather than on presentation and layout issues.

## 6.5 Multimedia

Multimedia usually stands for applications that handle different types of digital data originating from distinct types of media. The most common types of media in multimedia applications are text, sound, images, and video (which is an animated sequence of images). The digital data originating from each of these four

types of media is quite distinct in volume, format, and processing requirements (for instance, video and audio impose real time constraints on their processing). As an immediate consequence, different types of formats are necessary for storing each type of media.

In this section we cover formats and standard languages for multimedia applications. In contrast with text formats, most formats for multimedia are partially binary and hence can only be processed by a computer. Also, the presentation style is almost completely defined, perhaps with the exception of some spatial or temporal attributes.

### 6.5.1 Formats

Multimedia includes images, audio and video, as well as other binary data. We now briefly survey the main formats used for all these data types. They are used mainly in the Web and in digital libraries (see Chapters 13 and 15).

There are several formats for images. The simplest formats are direct representations of a bit-mapped (or pixel-based) display such as XBM, BMP, or PCX. However, those formats consume too much space. For example, a typical computer screen which uses 256 colors for each pixel might require more than 1 Mb (one megabyte) in storage just for describing the content of a single screen frame. In practice, images have a lot of redundancy and can be compressed efficiently. So, most popular image formats incorporate compression such as Compuserve's Graphic Interchange Format (GIF). GIF is good for black and white pictures, as well as pictures that have a small number of colors or gray levels (say 256). To improve compression ratios for higher resolutions, lossy compression was developed. That is, uncompressing a compressed image does not give the original. This is done by the Joint Photographic Experts Group (JPEG) format, which tries to eliminate parts of the image that have less impact on the human eye. This format is parametric, in the sense that the loss can be tuned.

Another common image format is the Tagged Image File Format (TIFF). This format is used to exchange documents between different applications and different computer platforms. TIFF has fields for metadata and also supports compression as well as different numbers of colors. Yet another format is Truevision Targa image file (TGA), which is associated with video game boards. There are many more image formats, many of them associated to particular applications ranging from fax (bi-level image formats such as JBIG) to fingerprints (highly accurate and compressed formats such as WSQ) and satellite images (large resolution and full-color images). In 1996 a new bit-mapped image format was proposed for the Internet: Portable Network Graphics (PNG). This format could be important in the future.

Audio must be digitalized first in order to be stored properly. The most common formats for small pieces of digital audio are AU, MIDI, and WAVE. MIDI is an standard format to interchange music between electronic instruments and computers. For audio libraries other formats are used such as RealAudio or CD formats.

There are several formats for animations or moving images (similar to video or TV), but here we mention only the most popular ones. The main one is MPEG (Moving Pictures Expert Group) which is related to JPEG. MPEG works by coding the changes with respect to a base image which is given at fixed intervals. In this way, MPEG profits from the temporal image redundancy that any video has. Higher quality is achieved by using more frames and higher resolution. MPEG specifies different compression levels, but usually not all the applications support all of them. This format also includes the audio signal associated with the video. Other video formats are AVI, FLI, and QuickTime. AVI may include compression (CinePac), as well as QuickTime, which was developed by Apple. As for MPEG, audio is also included.

### 6.5.2 Textual Images

A particular class of images that is very important in office systems, multimedia retrieval, and digital libraries are images of documents that contain mainly typed or typeset text. These are called *textual images* and are obtained by scanning the documents, usually for archiving purposes — a procedure that also makes the images (and their associated text) available to anyone through a computer network. The fact that a large portion of a textual image is text can be used for retrieval purposes and efficient compression.

Although we do not cover image compression in this chapter, we have seen that the most popular image formats include some form of compression embedded in them. In the case of textual images, further compression can be achieved by extracting the different text symbols or marks from the image, building a library of symbols for them, and representing each one (within the image) by a position in the library. As many symbols are repeated, the compression ratio is quite good. Although this technique is lossy (because the reconstructed image is not equal to the original), the reconstructed image can be read without problems. Additional information can be stored to reproduce the original image, but for most applications this is not needed. If the image contains non-textual information such as logos or signatures, which might be necessary to reproduce, they may be extracted through a segmentation process, stored, and compressed separately. When needed, the textual and non-textual parts of the image can be combined and displayed together.

Regarding the retrieval of textual images, several alternatives are possible as follows:

- At creation time or when added to the database, a set of keywords that describe the image is associated with it (for example, metadata can be used). Later, conventional text retrieval techniques can be applied to those keywords. This alternative is valid for any multimedia object.
- Use OCR to extract the text of the image. The resultant ASCII text can be used to extract keywords, as before, or as a full-text description of the

image. Depending on the document type, the OCR output could be reasonably good or actually quite bad (consider the first page of a newspaper, with several columns, different font types and sizes). In any case, many typos are introduced and a usual keyword-based query might miss many documents (in this case, an approximate search is better, but also slower).

- Use the symbols extracted from the images as basic units to combine image retrieval techniques (see Chapter 12) with sequence retrieval techniques (see Chapter 8). In this case, the query is transformed into a symbol sequence that has to match approximately another symbol sequence in the compressed image. This idea seems promising but has not been pursued yet.

### 6.5.3 Graphics and Virtual Reality

There are many formats proposed for three-dimensional graphics. Although this topic is not fully relevant to information retrieval, we include some information here for the sake of completeness. Our emphasis here is on the Web.

The Computer Graphics Metafile (CGM) standard (ISO 8632) is defined for the open interchange of structured graphical objects and their associated attributes. CGM specifies a two-dimensional data interchange standard which allows graphical data to be stored and exchanged between graphics devices, applications, and computer systems in a device-independent manner. It is a structured format that can represent vector graphics (for example, polylines or ellipses), raster graphics, and text. Although initially CGM was a vector graphics format, it has been extended to include raster capabilities and provides a very useful format for combined raster and vector images. A metafile is a collection of elements. These elements may be the geometric components of the picture, such as polyline or polygon; the appearance of these components; or how to interpret a particular metafile or a particular picture. The CGM standard specifies which elements are allowed to occur in which positions in a metafile.

The Virtual Reality Modeling Language (VRML, ISO/IEC 14772-1) is a file format for describing interactive 3D objects and worlds and is a subset of the Silicon Graphics OpenInventor file format. VRML is also intended to be a universal interchange format for integrated 3D graphics and multimedia. VRML may be used in a variety of application areas such as engineering and scientific visualization, multimedia presentations, entertainment and educational titles, Web pages, and shared virtual worlds. VRML has become the *de facto* standard modeling language for the Web.

### 6.5.4 HyTime

The Hypermedia/Time-based Structuring Language (HyTime) is a standard (ISO/IEC 10744) defined for multimedia documents markup. HyTime is an SGML architecture that specifies the generic hypermedia structure of documents.

Following the guiding principle of SGML, HyTime-defined structure is independent of any presentation of the encoded document. As an architecture, HyTime allows DTDs to be written for individual document models that use HyTime constructs, specifying how these document sets tailor the composition of these constructs for their particular representational needs. The standard also provides several metaDTDs, facilitating the design of new multimedia markup languages.

The hypermedia concepts directly represented by HyTime include

- complex locating of document objects,
- relationships (hyperlinks) between document objects, and
- numeric, measured associations between document objects.

The HyTime architecture has three parts: the base linking and addressing architecture, the scheduling architecture (derived from the base architecture), and the rendition architecture (which is an application of the scheduling architecture). The base architecture addresses the syntax and semantics of hyperlinks. For most simple hypermedia presentations, this should be enough. The scheduling module of HyTime defines the abstract representation of arbitrarily complex hypermedia structures, including music and interactive presentations. Its basic mechanism is a simple one: the sequencing of object containers along axes measured in temporal or spatial units. The rendition module is essentially an application of the scheduling architecture that defines a general mechanism for defining the creation of new schedules from existing schedules by applying special 'rendition rules' of different types.

HyTime does not directly specify graphical interfaces, user navigation, user interaction, or the placement of media on time lines and screen displays. These aspects of document processing are rendered from the HyTime constructs in a manner specified by mechanisms such as style sheets, as is done with SGML documents.

One application of HyTime, is the Standard Music Description Language (SMDL). SDML is an architecture for the representation of music information, either alone, or in conjunction with other media, also supporting multimedia time sequencing information. Another application is the Metafile for Interactive Documents (MID). MID is a common interchange structure, based on SGML and HyTime, that takes data from various authoring systems and structures it for display on dissimilar presentation systems, with minimal human intervention.

## 6.6 Trends and Research Issues

Many changes and proposals are happening, and very rapidly, in particular due to the advent of the Web. At this point, the reader must be lost in a salad of acronyms (we were too!), in spite of the fact that we have only mentioned the most important languages and formats. The most important of these are included in the Glossary at the end of this book. Some people believe that new

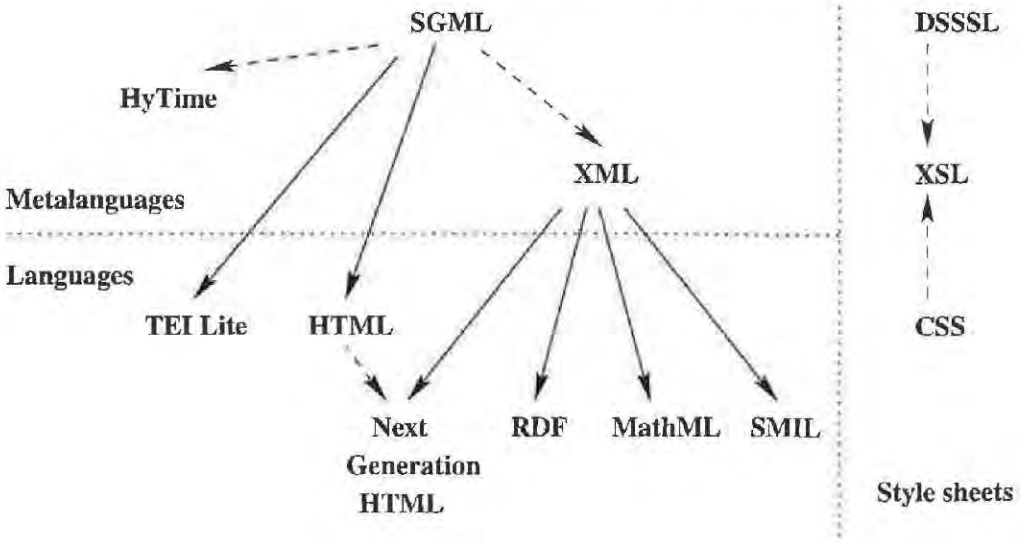



**Figure 6.6** Taxonomy of Web languages.

format specifications such as CSS or XML take away the simplicity of HTML, which is the basis of its success. Only the future will tell. Figure 6.6 illustrates a taxonomy of the main languages considered. Solid lines indicate instances of a metalanguage (for example, HTML is an instance of SGML), while dashed lines indicate derived languages. The main trend is the convergence and integration of the different efforts, the Web being the main application.

A European alternative to SGML is the Open Document Architecture (ODA) which is also a standard (ISO 8613 [398]). ODA was designed to share documents electronically without losing control over the content, structure, and layout of those documents. ODA defines a logical structure (like SGML), a layout and the content (including vector and raster graphics). An ODA file can also be formatted, processable, or formatted processable. Formatted files cannot be edited and have information about content and layout. The other two types can be edited. Processable files also have logical information in addition to content, while formatted processable files have everything. ODA is not used very much nowadays (see also Chapter 11).

Recent developments include:

- An object model is being defined: the document object model (DOM). DOM will provide an interoperable set of classes and methods to manipulate HTML and XML objects from programming languages such as Java.
- Integration between VRML and Dynamic HTML, providing a set of evolving features and architecture extensions to HTML and Web browsers that includes cascading style sheets and document object models.
- Integration between the Standard Exchange for Product Data format (STEP, ISO 10303) and SGML. STEP covers product data from a broad range of industries, and provides extensive support for modeling,

automated storage schema generation, life-cycle maintenance, and other management facilities.

- Efforts to convert MARC to SGML by defining a DTD, as well as converting MARC to XML. This has potential possibilities for enhanced access and navigation and presentation of MARC record data and the associated information.
- CGM has become of interest to Web researchers and commercial vendors for its use on the Internet, by developing a new encoding which can be parsed by XML.
- Several new proposals have appeared. Among them we can mention SDML (Signed Document Markup Language), VML (Vector Markup Language), and PGML (Precision Graphics Markup Language). The latter is based on the 2D imaging model of Postscript and PDF.

## 6.7 Bibliographic Discussion

The document model used in the introduction is based on [437]. Specific information on Web metadata is given in [487, 753]. Most of the information about markup languages and related issues is from the World Wide Web Consortium (see `www.w3.org`), in particular information on new developments such as DOM or SMIL. More information on SGML and XML is given by Goldfarb [303, 304]. Additional references in SGML are [369, 756] (in particular, the SGML example has been adapted from [24]). There are hundreds of books on HTML. Two sources for HTML 4.0 are [207, 796]. A book on CSS is [517]. For information on XML, XSL, and XLL see [795, 799, 798]. For a discussion about the advantages and disadvantages of XML and related languages see [182, 106, 455, 436]. More information on multimedia formats can be found in [501]. Formats for images and compression of textual images are covered in detail in [825].

# 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

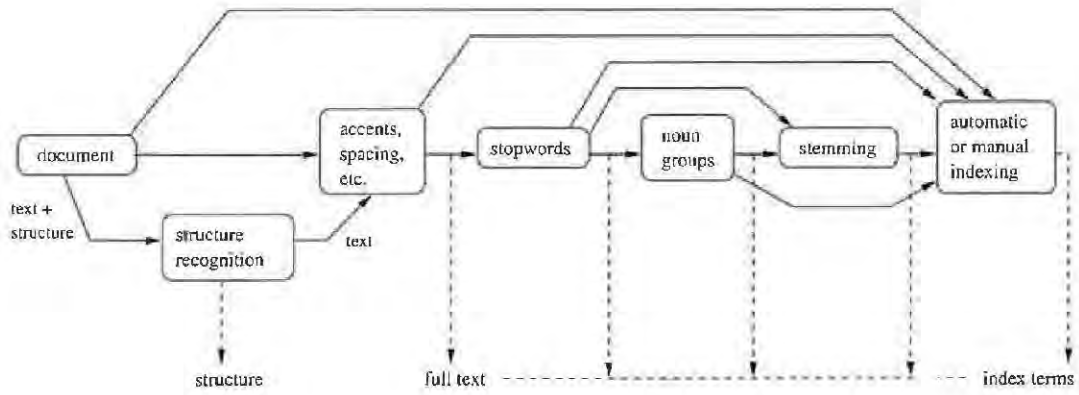



**Figure 7.1** Logical view of a document throughout the various phases of text preprocessing.

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 $\beta$ is a constant dependent on the particular text. For the 2 gigabyte TREC-3 collection [342], $\beta$ 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.

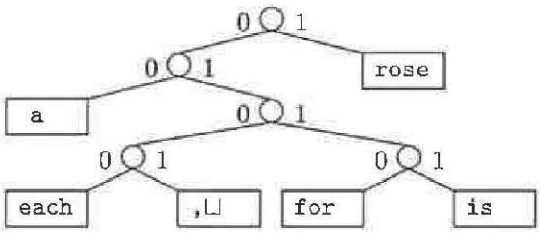
*Huffman Coding*

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`

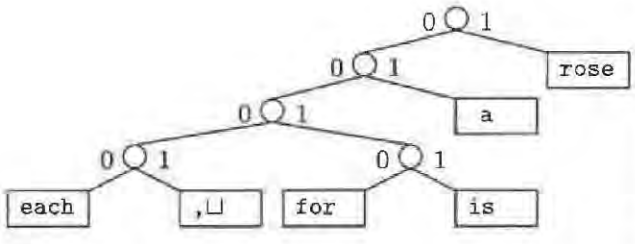
**Figure 7.2** Huffman coding tree for spaceless words.

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 $\ell$ 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.

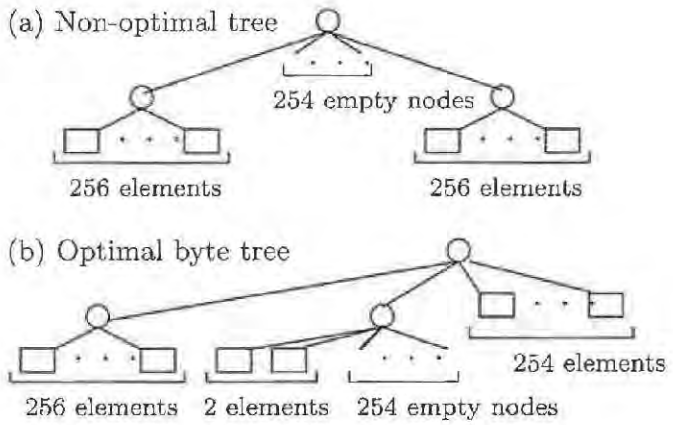



**Figure 7.4** Example of byte Huffman tree.

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-$\gamma$ code, which represents the number $x$ by a concatenation of two

| Gap $x$ | Unary | Elias-$\gamma$ | Elias-$\delta$ | 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-$\gamma$ codes are shown in Table 7.1.

The other coding scheme introduced by Elias is the Elias-$\delta$ code, which represents the prefix indicating the number of binary bits by the Elias-$\gamma$ code rather than the unary code. For $x = 5$, the first part is then 101 instead of 110. Thus, the Elias-$\delta$ codeword for $x = 5$ is 10101. In general, the Elias-$\delta$ 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-$\delta$ codes. In general, for small values of $x$ the Elias-$\gamma$ codes are shorter than the Elias-$\delta$ 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-$\gamma$ or Elias-$\delta$. 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-$\delta$, and Elias-$\gamma$, 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

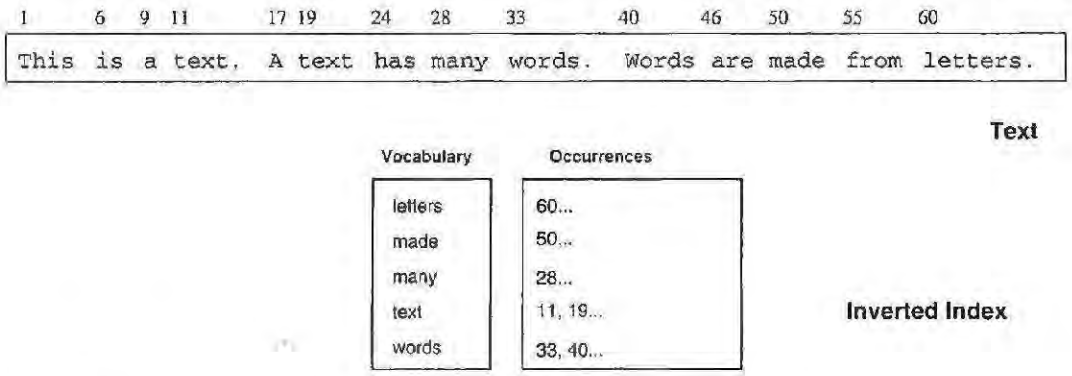



**Figure 8.1** A sample text and an inverted index built on it. The words are converted to lower-case and some are not indexed. The occurrences point to character positions in the text.

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 $\beta$ 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

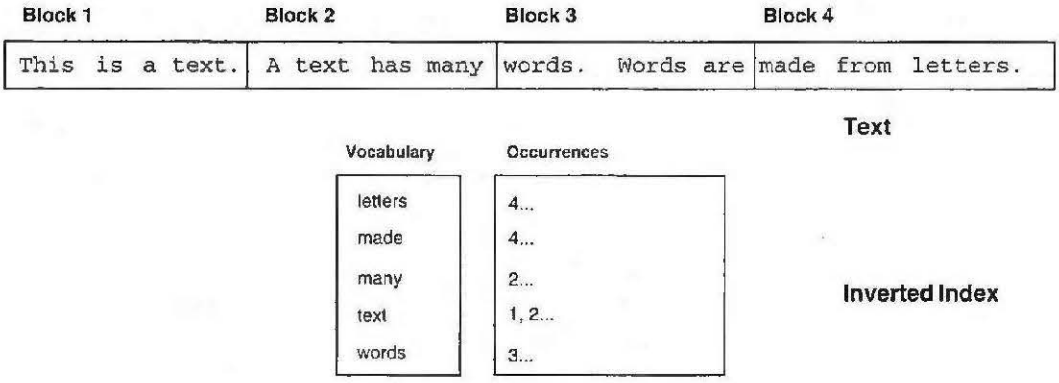



**Figure 8.2** The sample text split into four blocks, and an inverted index using block addressing built on it. The occurrences denote block numbers. Notice that both occurrences of 'words' collapsed into one.

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 $\alpha$ 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

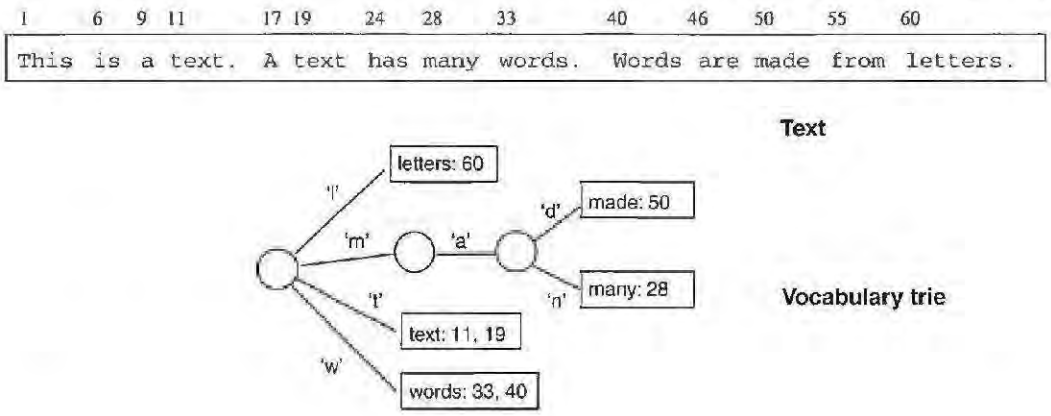



**Figure 8.3** Building an inverted index for the sample text.

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.

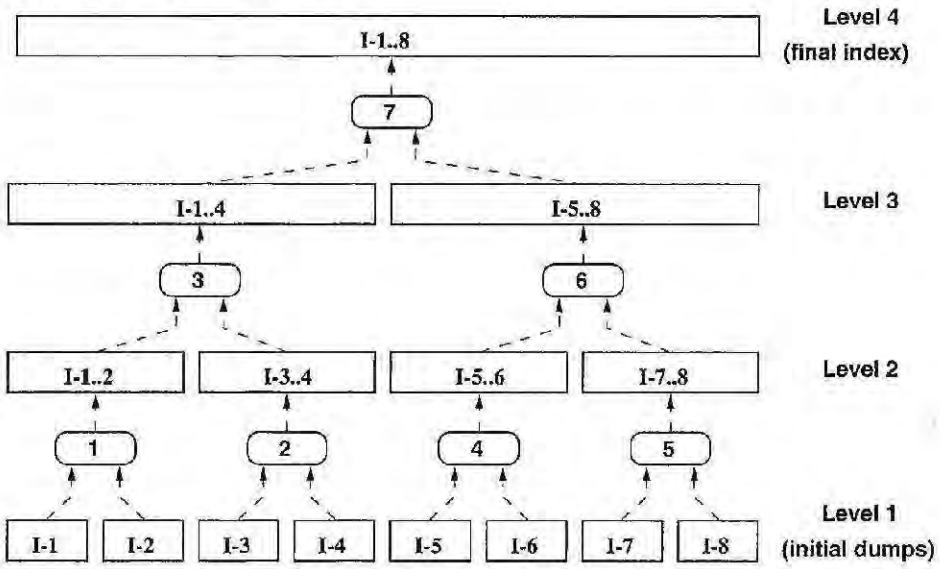



**Figure 8.4** Merging the partial indices in a binary fashion. Rectangles represent partial indices, while rounded rectangles represent merging operations. The numbers inside the merging operations show a possible merging order.

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).

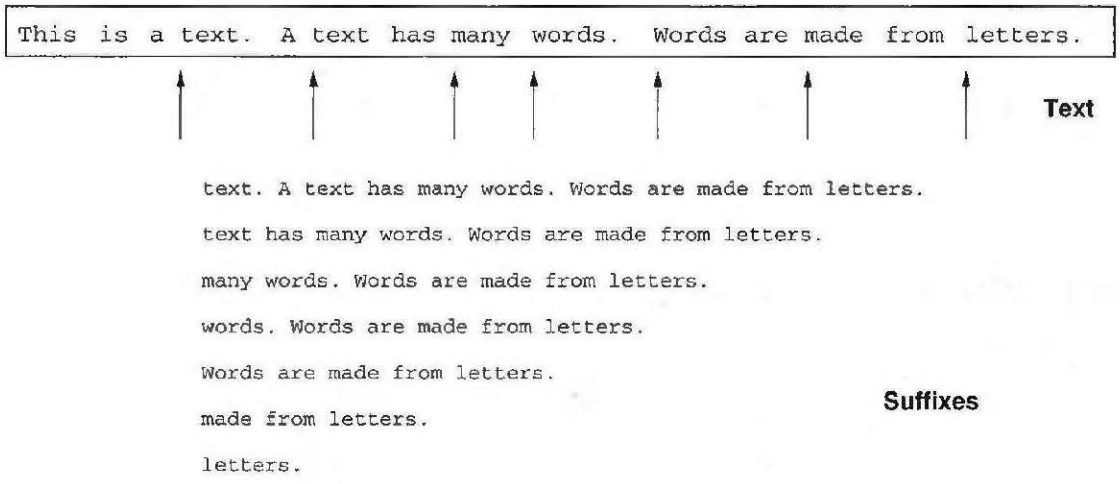



**Figure 8.5** The sample text with the index points of interest marked. Below, the suffixes corresponding to those index points.

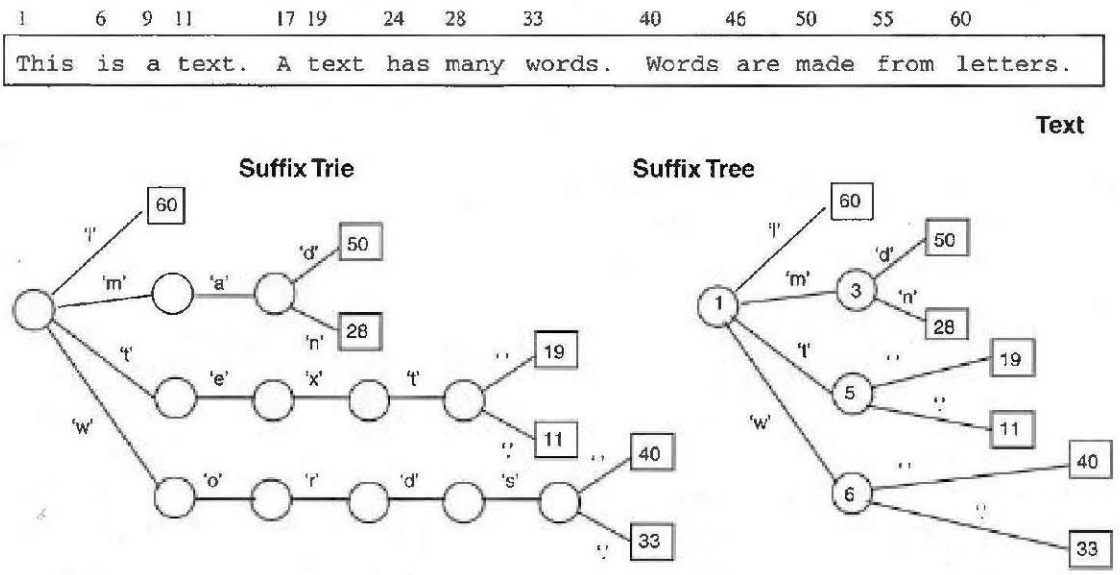



**Figure 8.6** The suffix trie and suffix tree for the sample text.

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

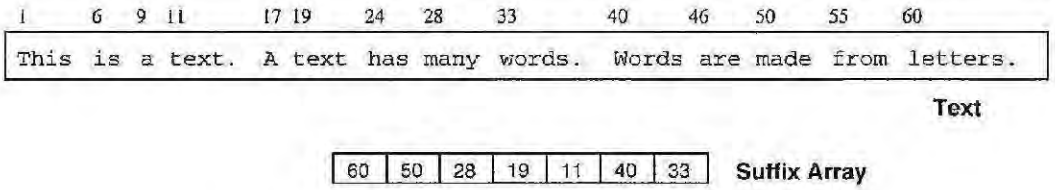



**Figure 8.7** The suffix array for the sample text.

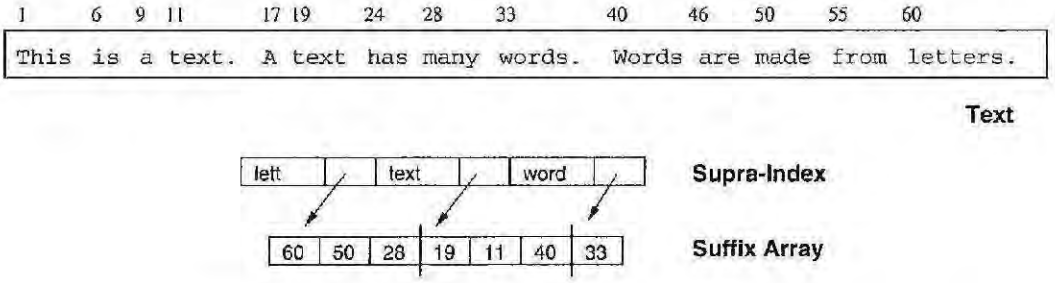



**Figure 8.8** A supra-index over our suffix array. One out of three entries are sampled, keeping their first four characters. The pointers (arrows) are in fact unnecessary.

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 $\ell$ 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 $\ell$ 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).

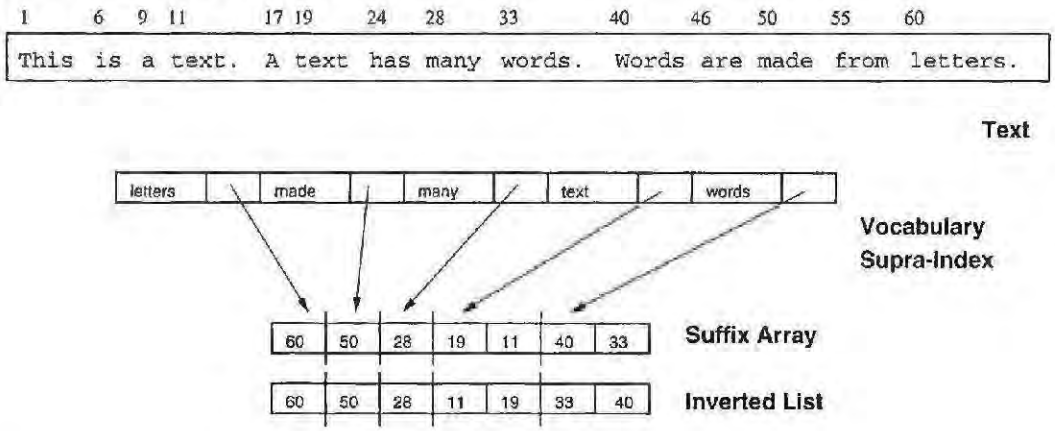



**Figure 8.9** Relationship between our inverted list and suffix array with vocabulary supra-index.

## 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 $T_{a\ldots}$ and $T_{b\ldots}$ 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,