Information Retrieval in Digital Libraries: efficient catalog searches using tries

Alejandro Bia
"Miguel de Cervantes" Digital Library
University of Alicante
E-03071 Alicante, Spain
abia@dlsi.ua.es

Alfonso Nieto
"Miguel de Cervantes" Digital Library
University of Alicante
E-03071 Alicante, Spain
fox@bibvirtual.bimicesa.ua.es

ABSTRACT

Digital Libraries of literary works usually store a huge amount of textual information. It is obvious that the mere accumulation of texts leads only to a limited-use library. Hence the need for efficient information retrieval services. The use of indices to speed up the search is advisable in cases like ours, the "Miguel de Cervantes" digital library, where the text collection is relatively large (4000 books at present) and semi-static (updated at reasonably large intervals).

We developed a catalog search engine based on tries that performs fast searches with efficient results. However the size of the trie structures is quite big, proving adequate for catalog searches, but not for whole text indexing.

Keywords

advanced data structures, information retrieval, object-oriented databases, text markup, trie

1. INTRODUCTION

"Digital library: the combination of a collection of digital objects (repository); descriptions of those objects (metadata); a set of users (patrons or target audience or users); and systems that offer a variety of services such as capture, ***indexing, cataloging, search****, browsing,* ***retrieval****, delivery, archiving, and preservation. (From Modern Information Retrieval, Glossary [1])"*

A deep study about a book or an author surely starts with a catalog search. Searches can be performed sequentially over the proper text files, but obtaining the results in this way would require an enormous computing time because the computer has to scan all the files implied in the query looking for a keyword or a whole sentence embedded. A sequential or online search would only be advisable when the text collection is really small and the texts are volatile, or when the space overhead for indices cannot be afforded.

The use of indices to speed up the search is advisable in cases where the text collection is large and semi-static (updated at reasonably regular intervals) [1]. Our catalogs fall within this category since they are updated weekly. Baeza-Yates and Ribeiro-Neto also suggest three main indexing techniques for these cases: inverted files, suffix arrays and signature files, and highlight the advantages of the first two.

Therefore, if the goal of our DL is not only to publish d-books but also to provide advanced searching tools to the researchers it is necessary to store the books using data structures that allow the computer to query documents faster. The best solution is a search engine based on reverse indexes (hereafter, TRIE) loaded with books marked up using a structural markup language such as SGML or XML.

2. INVERTED INDEXES SEARCH ENGINE

As Rijsbergen [16] stated in his classic book about "Information Retrieval": More recently a special kind of tree, called a trie, has attracted attention. This is a tree structure which has records stored at its terminal nodes, and discriminators at the internal nodes. A discriminator at a node is made up from the attributes of the records dominated by that node.

As Knuth [11] puts it: "A trie is essentially a M-ary tree whose nodes are M-place vectors with components corresponding to digits or characters. Each node on level l represents the set of all keys that begin with a certain sequence of l characters; the node specifies an M-way branch depending on the (l + 1)st character." Tries were invented by Fredkin [10], further considered by Sussenguth [15], and more recently studied by Burkhard [5], Rivest [13], and Bentley [2]. The use of tries in data retrieval where one is interested in either a match or mismatch is very similar to the construction of hierarchical document classification, where each node of the tree representing the hierarchy is also associated with a 'discriminator' used to direct the search for relevant documents."

So a trie (reTRIEval) is an advanced data structure used to store and look for strings (strings of characters, integers, etc., strings of words in our case), with a linear temporary complexity [9], i.e., they are able to retrieve any string in a time proportional to its length [1]. All the more so, the response time to search one string on a trie does not depend on the number of strings it holds. This behavior is really useful when managing huge amounts of strings.

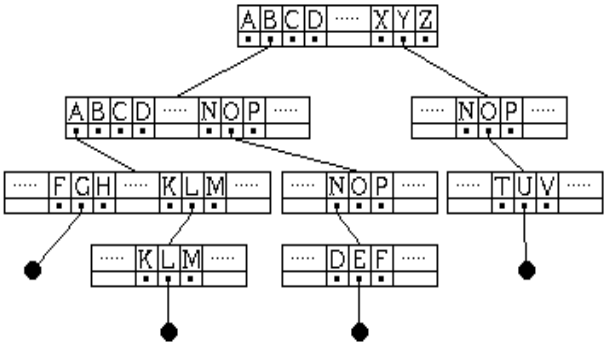

Figure 1: A simple trie example.

The structure of a trie is quite similar to the structure of a tree. The usual node of a trie of letters may have up to 26 subnodes, one for each letter of the alphabet (27 in our case, including the Spanish "ñ"). One path from the root of the trie to one leaf represents one string in the trie. The pointer of each node used in that path represents one character of the string. Let's see this with a brief example.

The trie of figure 1 holds the words BAG, BALL, BONE and YOU. The words that start with the same letters share nodes of the trie, achieving an important compression rate. In many languages, like in Spanish, words are often made up of prefixes, roots and suffixes, specially verb forms, achieving high levels of overlapping in a trie structure. Black dots in the figure represent string ends. At the end of a string, leaf nodes point to objects holding information about the string found in the trie (could be for instance the definition of that word, a numerical key to encode the word, or a pointer to another structure).

In spite of allowing for fast searches, this vast structure requires a huge amount of memory as the number of strings grows, so it is necessary to compress the structure even more, by means of some optimizations explained later, and to make it persistent, using object-oriented databases as the best solution.

3. MEMORY OPTIMIZATIONS

In the example, the trie holds the word YOU, which does not share any letter with the rest of words in the trie. In that case, it is possible to change the nodes corresponding to letters **O** and **U** by a simple substring **OU** which finally points to the information object corresponding to the word YOU. This method is called suffix compression.

In the same way, it is also possible to change intermediate nodes by other substrings. For example, a trie with the words "accommodate" and "accommodation" can be represented with a root node, which would point (with its first pointer **A**) to a substring **CCOMMODAT**. Then, this substring would point to another node that would point to two different suffixes ("E", "ION"); each one would finally point to its information object.

The trie is a structure that normally store words (as a dictionary) but, in the Miguel de Cervantes DL [4, 3], we are using it to store whole phrases (book titles, author names, subjects and dates) i.e. everything that can be used as a keyword in a catalog search. As a sentence is a sequence of words (a string of words) and white spaces, it is possible to store them in a different way. First we store the words in one trie (WORDS trie) and assign one identifier to each different word, usually smaller than the word itself. Then, in a second trie (MAIN trie), we store strings of identifiers, instead of the strings of words, achieving a considerable reduction in storage space. This is called a multi-tier trie. With this strategy, we do not only make use of common-letters compression but also common-words compression. [1]

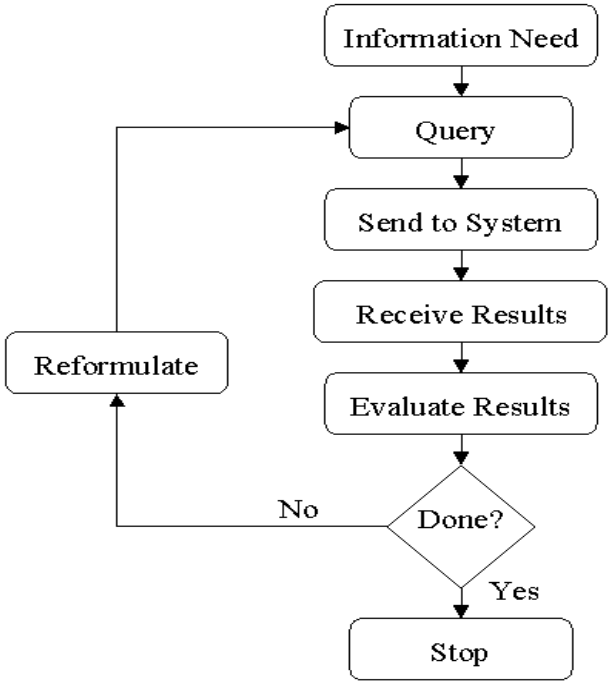

Figure 2: Information access model used.

4. OBJECT-ORIENTED DATABASES

Using this multi-tier trie technique, we can create a trie in memory that can hold the whole catalog of books, authors and subjects but that trie is not persistent. If the computer is powered down or the system crashes, we should create the tries all over again by reading and parsing all the catalogue information. An alternative we use is to create them only once and save them in an object-oriented database. Object-oriented databases allow programmers to save data structures from memory to disk in a transparent manner. Furthermore, they can also load from disk to memory only the parts of the trie most commonly used. Thanks to this behavior it is possible to manage relatively fast tries of sizes bigger than the available memory.

[1] Depending on the language, we could build three or four-tier tries if the vocabulary of that language has lots of prefixes and suffixes.

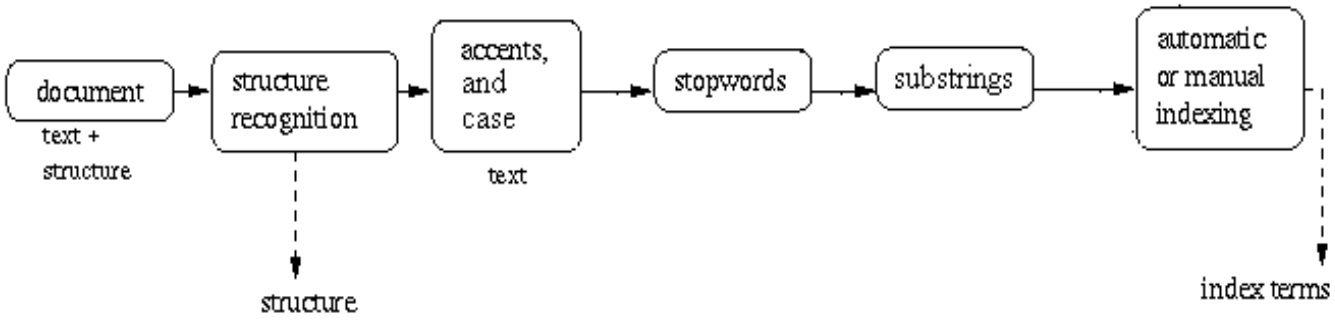

Figure 3: Extraction of relevant keywords for the index.

5. SEARCH KEYWORDS

Once we had the search engine ready, we had to study what to store in it, in other words, the keywords the search engine will have to find and the information it should return.

Standard catalog queries are about authors, book titles, subjects and dates. Keywords for these queries could be full or partial names of authors, nicknames of authors, partial titles of books, secondary titles, secondary author names, dates of birth and death of the author, starting and ending writing dates of a book, or just the century, subjects (poetry, history, geography), etc.

The problem is that the users often do not know exactly the full names of the authors or the complete title of a book. To avoid this problem, our approach was to store all possible substrings of the original author name or title strings. Irrelevant words (stopwords) such as articles, prepositions, and other connectives) which do not add much information, are not included in the trie structure, and are ignored in user queries. Accented and uppercase letters are converted to non-accented lowercase and ignored at query time as well (see figure 3).

If we look for "La ilusión", for instance, the system will return all the information objects whose keywords or subkeywords start with "ilusion". If we want to store the book title "El Ingenioso Hidalgo Don Quijote de La Mancha", we will store (after lowercase-accents conversion and stopwords elimination): "ingenioso hidalgo quijote mancha", "hidalgo quijote mancha", "quijote mancha" and "mancha". All these keywords will finally point to the same information object, the famous Cervantes masterpiece.

6. INFORMATION OBJECTS

If we expect the user to be able to find pieces of information from within the books, we have to be able to detect and store every relevant piece of information such as titles, author names, subjects, dates, etc. in the trie structure. Here is where the use of a structural markup is important. The combination of XML [18, 17, 7] and the TEI [14, 12, 6] allows a parser to recognize the relevant pieces of text that must be stored in the trie as keywords for further searches. If we know that the phrase "La vida es sueño" is a book title, it is possible to store it as a search keyword. This keyword will point to an information object, which will refer to the object on the Document Object Model (DOM) [8] associated with that title [2].

The reason why we store references to DOM objects is simple. When we look for a keyword, we can obtain many information objects which hold semantic data used to filter the search. For example, if we are looking for an author, we should only retain the objects that belong to authors. On the contrary, if we are looking for book titles, we should refuse the information objects that do not belong to this category.

7. CONCLUSIONS

We developed a catalog search engine that performs fast searches with efficient results. However the size of the trie structures is quite big, being adequate for catalog searches, but not for whole text indexing in a Digital Library.

With about 4000 books, the object-oriented database file for the *main trie* using a single trie approach and storing keywords for book titles, authors, subjects and dates occupied approximately 60 Mb. Using a multi-tier trie scheme, with a *word trie* for numerical coding of the words the size of the *main trie* is reduced to about one third of its original size (20 Mb). Compressing the *main trie* file with a standard file compressor we achieved a compression rate of 0.26 for comparison.

Concerning user-machine interaction, we have used the standard information access model used by most Web search engines, as described by [1] (see figure 2). This is a stepwise refinement model where a query is submitted to the search engine, processed for results, evaluated by the user, and refined in case of unsatisfactory results.

An HTML form to test or use this catalog-search engine can be found in: http://cervantesvirtual.com/busquedas.shtml (see figure 4)

8. ACKNOWLEDGMENTS

Thanks to Pedro Pastor for some of the ideas concerning the development of this search engine.

[2]The search engine was designed following the object oriented paradigm and implemented in Java using Objecstore for data persistance

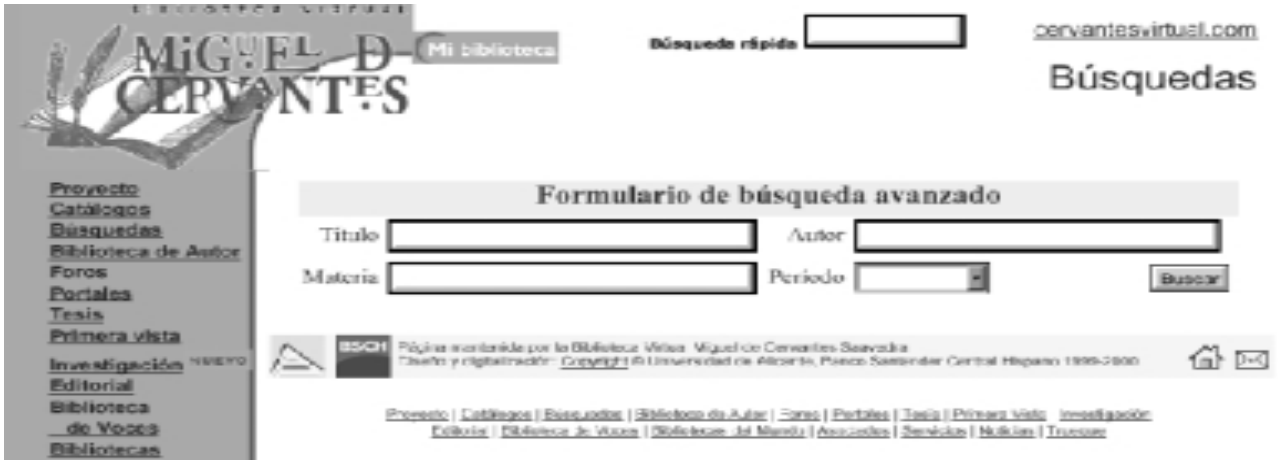

Figure 4: Catalog search form of the Miguel de Cervantes DL.

9. REFERENCES

[1] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. ACM press and Addison Wesley, Edinburgh Gate, Harlow, Essex CM20 2JE, England, 1st edition, 1999. See also http://www.dcc.ufmg.br/irbook or http://sunsite.dcc.uchile.cl/irbook.

[2] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 13:675–677, 1975.

[3] A. Bia and A. Pedreño. The Miguel de Cervantes Digital Library: The Hispanic Voice on the WEB. *LLC (Literary and Linguistic Computing) journal, Oxford University Press*, (to be published soon) 2000. Presented at ALLC/ACH 2000, The Joint International Conference of the Association for Literary and Linguistic Computing and the Association for Computers and the humanities, 21/25 July 2000, University of Glasgow.

[4] Biblioteca Virtual Miguel de Cervantes Saavedra. http://cervantesvirtual.com, 1999.

[5] W. A. Burkhard. Partial match queries and file designs. In *Proceedings of the International Conference on Very Large Data Bases*, pages 523–525, 1975.

[6] L. Burnard. Text encoding for information interchange: An introduction to the text encoding initiative. http://www-tei.uic.edu/orgs/tei/info/teij31/index.html, 7 1995.

[7] R. Cover. XML at OASIS. http://www.oasis-open.org/cover/xml.html.

[8] DOM, Document Object Model (W3C). http://www.w3.org/DOM.

[9] A. Ferrández et al. *Tipos Abstractos de Datos*. University of Alicante, 1997.

[10] E. Fredkin. Trie memory. *Communications of the ACM*, 3:490–499, 1960.

[11] D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, volume 1 of *The Art of Computer Programming series*. Addison Wesley, Reading, Massachusetts, 3rd edition, July 1997. (first edition 1973).

[12] W. Plotkin and C. Sperberg-McQueen. Text Encoding Initiative, homepage (tei@uic.edu). http://www.uic.edu/orgs/tei/index.html.

[13] R. Rivest. *Analysis of Associative Retrieval Algorithms*. Computer science, Stanford University, 1974.

[14] C. M. Sperberg-McQueen and L. Burnard, editors. *Guidelines for Electronic Text Encoding and Interchange (Text Encoding Initiative P3), Revised Reprint, Oxford, May 1999*. TEI P3 Text Encoding Initiative, Chicago - Oxford, May 1994.

[15] E. H. Sussenguth. Use of tree structures for processing files. *Communications of the ACM*, 6:272–279, 1963.

[16] C. J. van Rijsbergen. *Information Retrieval*. Butterworths, London, 2nd edition, 1979. See also: http://www.dcs.glasgow.ac.uk/Keith/Preface.html.

[17] World Wide Web Consortium (W3C). http://www.w3.org/.

[18] XML, Extensible Markup Language (W3C). http://www.w3.org/XML.