TR-93-003

A Two-Level Approach to Information Retrieval

Udi Manber and Sun Wu

A Two-Level Approach to Information Retrieval

(Preliminary Version)

Udi Manber[1]

Department of Computer Science
University of Arizona
Tucson, AZ 85721

and Sun Wu

Institute of Information Science
Academia Sinica
Taipei, Taiwan, ROC.

January 1993

ABSTRACT

A new indexing and query schemes for information retrieval of medium-size natural language text are presented in this paper. The novelty of the algorithms is that they use a very small index — in most cases 2-4% of the size of the text — and still allow very flexible full-text retrieval including the usual Boolean queries but also approximate matching. The ability to perform approximate queries — to search for misspelled keywords — is very powerful. Query times are typically slower than with inverted files, but they are still fast enough for many applications. We also describe a prototype system, called a *Personal Information Retrieval System (PIRS)* that we developed based on the new algorithms. Although the algorithms we present are general, PIRS was especially designed for personal information as opposed to typical IR systems that are designed for central collections used by many people. By personal information we mean information generated and collected by single users for their purposes. It can include personal correspondence, articles of interest, e-mail messages, personal notes, bibliographic files, etc. The main characteristic of such information is that it is very non-uniform and includes many types of documents. An IR system for personal information should support low overhead, many types of queries, flexible interaction, and customization, all of which are important features of PIRS.

---

[1] Supported in part by an NSF Presidential Young Investigator Award (grant DCR-8451397), with matching funds from AT&T, and by an NSF grant CCR-9001619. Part of this work was done while the author was visiting the University of Washington.

1. Introduction

The most common data structure used in information retrieval (IR) systems is an inverted list [SM83]. All occurrences of each word are stored in a table indexed by that word (using a hash table). The only exception are common words, such as *the* and *that*, which belong to a *stop list* of words that are not indexed. Inverted lists allow very fast queries: There is no need to search in any of the texts, only the table needs to be consulted and the places where the word occurs are retrieved immediately. Boolean queries and proximity queries are slower, but are still relatively fast.

One drawback of inverted lists is their space requirement. The size of the index is usually in the same order of magnitude as the size of the text. This may not be a major drawback for commercial text databases, because disk space is relatively cheap, but it is a major drawback for personal information. Most users would not agree to double their disk cost for the benefit of indexing. Indeed today most personal file systems are not indexed. But, due to an increased availability of digital information through networks, many personal file systems are large enough to require IR capabilities. It is very common to forget the location of some information and not be able to find it. Signatures files [Fa85, GB91] have been suggested as an alternative to inverted indexes. Their indexes are only 10%-30% of the text size, but their search time is slower, and since they are based on hashing their parameters must be chosen carefully (especially for many files of different sizes) to minimize the false drops probability.

The second weakness of inverted lists, and in fact of all other suggested IR schemes that we are familiar with, is the need for exact spelling. If one is looking for all articles containing Schwarzkopf for example, any article with a misspelling will be missed; not to mention that one needs to find the exact spelling to form the query. The only way to find misspelled words in the text is to try different possibilities by hand, which is frustrating, time consuming, and there is no guarantee of success. This problem is expected to become more acute as more information is being scanned by OCR (Optical Character Recognition) devices which currently have an error rate of 2-5% [BN91, RKN92].

The scheme presented in this paper overcomes both weaknesses of inverted lists. It requires a very small index, in most cases 2-4% of the original text, and it supports arbitrary approximate matching. The price we pay is speed. Our algorithms are slower that ones using inverted lists, but their speed is still in the order of a few seconds, which is fast enough for single users. For some applications, such as management of personal information, speed is a secondary issue. Most users would rather wait for 10-15 seconds for a query than double their disk space. Even for IR systems, such as library card catalogs, where high throughput is essential, our scheme can be used as a secondary mechanism to catch spelling errors. We found several spelling errors in our library catalog experiment (section 3.3). We believe that this capability is essential in all applications that require a high level of reliability. For example, medical labs can miss information on patients due to misspelling [MS93].

We call our method *two-level searching*. The idea is a hybrid between full inverted lists and sequential search with no indexing. It is based on the observation that with current computing performance sequential search is fast enough for text of size up to several megabytes. Until several years ago, sequential search of, say a 1MB file, took a minute or more of running time. Therefore, inverted lists were essential. Today, on a SUN SparcStation for example, using a fast search such as our *agrep* program [WM92], we can search a 1MB file in less than a second. Therefore, there is no need to index every word with an exact location. In the two-level scheme the index does not provide exact locations, but only pointers to a reasonably small area where the answer may be found. Then, a flexible sequential search is used to find the exact answer and present it to the user. (Some other IR systems, such as MEDLARS [SM83], allow sequential search as a postprocessing step to further filter the output of a query, but the search relies entirely on inverted lists.) Furthermore, the index itself, since it is reasonably small, can be searched with sequential search, rather then by hashing methods. This provides additional flexibility, in particular it allows approximate matching, as we will discuss later.

Due to the need for some sequential search, the 2-level scheme, as it stands right now, is not suitable for very large text collections. We found it very effective for up to 100 MBytes of text, and less so for 250MB. We expect it to be too slow for over 500MB with current technology. As CPU and I/O speeds increase, so would the limit on the size of the texts. This is especially true for parallel computers which the two-level scheme can use very well (by having different processors search different blocks). It is not unreasonable to expect to be able to handle one GBytes of text in 2-3 years without specialized hardware. Some applications will no doubt grow in size just as fast as computing power grows, but not all. There are quite a few applications that are inherently limited in size, or have growth patterns that are significantly slower than the expected growth of computing power. Such applications include various types of information about people (population growth is much slower than computing power growth), geography (e.g., description of travel destinations, streets, and sites), and personal notes. For those applications, our two-level approach is attractive because it requires much less space and it allows more flexible queries.

The two-level approach is described in detail in the next section. Section 3 describes PIRS, the personal information retrieval system that we built based on the two-level approach. We also discuss general issues in the design of personal IR systems, and present some preliminary experiments with PIRS. Section 4 concludes with suggestions for future work.

2. The Two-Level Query Approach

In this section, we describe our scheme for two-level indexing and searching. We start with the way the index is built.

The information space is assumed to be a collection of unstructured text files. A text consists of a sequence of *words*, separated by the usual delimiters (e.g., space, end-of-line, period, comma). The first part of the indexing process is to divide the whole collection into smaller

pieces, which we call *blocks*. We try to divide evenly so that all blocks have approximately the same size, but this is not essential. The only constraint we impose is that the number of blocks does not exceed $2^8 = 256$, because that allows us to address a block with 8 bits or one byte. Again, this is not essential, but it appears to be a good design decision.

We scan the whole collection, word by word, and build an index that is similar in nature to a regular inverted index with one notable exception. In a regular inverted index, every occurrence of every word is indexed with a pointer to the exact location of the occurrence. In our scheme every word is indexed, but not every occurrence. Each entry in the index contains a word and the block numbers in which that word occurs. Even if a word appears many times in one block, only the block number appears in the index and only once. Since each block can be identified with one byte, and many occurrences of the same word are combined in the index into one entry, the index is quite small. Full inverted indexes must allocate at least one word (4 bytes), and usually slightly more, for each occurrence of each word. Therefore, the size of an inverted index is comparable to the size of the text. But our index contains only the list of all unique words followed by the list of blocks — one byte for each — containing each word. For natural language texts, the total number of unique words is not too large and quite limited regardless of the size of the text. Experiments with different types of texts are discussed in section 3.3.

The search routine consists of two phases. First we search the index for a list of all blocks that may contain a match to the query. Then, we search each such block separately. Both phases are done with sequential search using *agrep*, the pattern-matching program that we developed. Agrep includes several new pattern-matching algorithms and it is described in detail in [WM92]; we only briefly list its main features here. Agrep can quickly search for many types of patterns including ones with wild cards, with classes of characters, with complements, and even arbitrary regular expressions. A search can be exact or approximate. Substitutions, insertions, and/or deletions can be allowed, and the relative weight of each of them can be changed. Agrep can search for all matches within a specified number of errors, or find the matches with the minimal number of errors. Patterns can include parts that must match exactly and parts that can have errors. Boolean queries are supported. Matches are made to either lines (the default) or to user-defined *records* (e.g., paragraphs, e-mail messages, or whole files). Agrep's source code is available by anonymous ftp from cs.arizona.edu (as well as from 25 other ftp servers).

Even though the first phase can be done by hashing, we prefer sequential search because it allows us to perform approximate queries, subword queries (even a subword that occurs in the middle of the word), and other complicated queries (e.g., containing wild cards). Using hashing, the only keywords you get are those that were selected to be hashed. With sequential search we can get the full power of agrep. Since the index is quite small we can afford sequential search.

Boolean queries are performed in a similar way to the regular inverted lists algorithm. Suppose the query is for *pattern1* AND *pattern2*. We find the list of all blocks containing each pattern and intersect them. Then we search the blocks in the intersection using agrep's efficient Boolean query algorithm (which scans the text only once and searches for all parts of the query

concurrently). Notice that even if we find some blocks that contain *pattern1* and *pattern2*, it does not mean that the query is successful, because *pattern1* may be in one part of the block and *pattern2* in another.

Approximate queries are handled by first using agrep to find all words in the index that match approximately with the pattern. Then the corresponding blocks are searched, using agrep again, to find the particular matches. The same procedure holds for other complicated patterns such as ones that contain wild cards (e.g., Model2...Z), a set of characters (e.g. AB[0-9]CD, where [0-9] stands for any digit), or a negation (e.g., [^0-9][A-D]END, which stands for a word whose first character is not a digit, its second character is A, B, C, or D, and the next characters are E, N, and D). These kinds of patterns cannot be supported by the regular hashing scheme that looks up a keyword in the table, because such patterns can correspond to hundreds or even thousands of possible keywords.

The main weakness of our scheme is that some queries may match many blocks requiring a sequential search of a large portion of the information space. One common reason for such queries is that they are not specific enough, in which case any scheme will lead to a huge output. The only time our scheme is unusually inefficient is when there are only a few matches to the query, but they are dispersed among the blocks. We believe that this is not common, because the blocks usually contain related pieces of text. For example, if the text is a library catalog the blocks correspond to ranges of call numbers. A typical query is limited to a few call numbers, thus it will involve only few blocks. Few queries will have matches in many different call numbers and only few matches for each. One way to minimize this problem is to use *clustering techniques* [JD88] when we partition the information space. We have not yet implemented any clustering algorithms for PIRS, and have not yet studied them sufficiently. Another possible option is a warning to the user that a query match many blocks and therefore its completion may require a longer delay. In that case, the user can either refine the query or check the list of matched blocks and select the ones that seem more promising. We can determine the identity of the potential matches directly from the index without having to do extensive sequential search.

In summary, we list the strengths and weaknesses of our two-level scheme compared with the regular inverted lists:

Strengths

1. Very small index.
2. Approximate matching is supported.
3. Easy to modify the index due to its small size. Therefore, dynamic texts can be supported.
4. No need to define document boundaries ahead of time. It can be done at query time.
5. Easy to customize to user preferences.
6. Easy to adapt to a parallel computer (different blocks can be searched by different processors).

7. No need to extract stems. Subword queries are supported automatically (even subwords that appear in the middle).
8. Queries with wild cards, classes of characters, and even regular expressions are partially supported.

Weaknesses

1. Slower compared to inverted lists for some queries. Not suitable for applications where speed is the predominant concern.
2. Too slow, at this stage, for very large texts (more than 500MB).
3. Speed is not uniform. In some bad cases, a query can take a very long time.
4. Requires extra work, when the index is built, to partition the information space.

3. PIRS — A Personal Information Retrieval System

3.1. The Need for Personal IR Systems

There are a large number of programs and small devices called *personal information managers* or similar names. These are typically designed for very specific tasks, such as scheduling, calendar, limited amount of personal notes, etc. They are not suitable for large amount of information. There are also several software packages, mostly for personal computers, that search hard disks for information. The ones we experimented with did not use an index; they essentially search the disk sequentially taking a lot of time for disks containing a large amount of text files.

We are not familiar with work dealing with personal information management in the context of unstructured large number of text files, which leads to the obvious question. Why do we need personal IR systems? Can't people organize their information so that they know where things are? Organizing one's data has always been a hard problem for some people, and with the growth of available information it will soon be too hard for almost everyone. Below we list five reasons for the need of personal IR systems:

1. Until recently computers were used only for work purposes and mostly for specific tasks, but their use is now ubiquitous. Information of interest includes many topics and in many cases there is significant overlap, so that storing a piece of information in one place precludes finding it when it is relevant to another topic. For example, e-mail messages may contain personal and professional information in the same message; a topic that belongs to two (or more) different projects may be put in the area for one of them; a new topic may originally be filed under miscellaneous, and only later be important (where did I put the notes of that interesting colloquium two years ago?).
2. Memory fades with time. It's hard, even with a good organization, to remember things from a few years back. After a while, not only the content of a piece of information can be forgotten, but also the *existence* of that information. In only the first week of experimenting

with PIRS one of us found pertinent information related to a current project that was written (in a different context) several years ago and forgotten.

3. The size, and more important the complexity of information is growing too fast.
4. Some people are messy. They need help too.
5. People have access to information that was gathered by other people who are not necessarily librarians. You don't know the logic behind their organization of the data, and even when you do, you usually want to adapt it to your sense of it.

These reasons and others have motivated extensive research in several areas including cognitive science, database design, and hypertext systems. A personal IR system as we suggest here will not solve the problems by itself, but it can be of great help. Most people currently do not have too much "personal information" to maintain and thus do not require sophisticated search facilities. But with cheap disk cost, easy access to networks, on-line services, increased e-mail usage, and other technological advances in the dissemination of information, personal IR systems will be essential.

3.2. The Current Prototype of PIRS

PIRS is written in C under the UNIX operating system. It is a collection of many separate programs of a total of about 7500 lines of code. The current implementation is geared towards indexing a part of the file system. The user can indicate which directories should be indexed or can simply index everything in the home directory (which will include going down the tree to all the files). Before indexing a file, the program checks whether it is indeed a text file. If the file is found to have too many non-alphanumeric characters (e.g., an executable or a compressed file), it is not indexed. Other formats are excluded too. For example, 'uuencoded' files and 'binhexed' files[2] are excluded. Determining whether or not a file is a real text file is not easy. Texts of languages that use special symbols, such as an umlaut, may look like binary files. There are tools that do a good job of identifying the type of file, for example [HS93], and we will eventually incorporate them in PIRS. On the other hand, some files that are text files for all syntactic purposes should not be indexed. A good example is a file containing DNA sequences. We actually have such files (we do research on pattern matching in DNA) and found them to cause the index to grow significantly, because they contain a large number of essentially random words. We should note that the two-level scheme is not suitable for typical biological applications, because those require more complicated types of approximate matching. Another example are files that contain mainly numeric information. Initially, we did not index numeric "words," but found the ability to use dates and other identifying numbers to be very useful (e.g., find all e-mail messages from a certain person during July 1991). We currently allow the user to choose whether or not to index numbers. But files with numeric data will make the index large unnecessarily. One possible

[2]uuencode and binhqx are programs that translate binary files into an ASCII file for transmission through e-mail.

solution is to identify the files that contribute too many words to the index, exclude these files from the index, and notify the user so that he/she can add these files.

The partition into blocks is currently done in a straightforward way. The total size of all text files is computed, and an estimate on the desired size of a block is derived. Files are then combined until they reach that size, and a new block is started. We plan to improve this scheme in the future in two ways: 1) the partition should be better adapted to the original organization of the data, and 2) the user should have the ability to control how the partition is done.

The user interface we currently employ is identical to that of agrep, except that no file name is given by the user. So, whereas `agrep information file_name` will output all lines containing *information* in the file *file_name*, `find information` will output all lines *in all indexed files* that contain *information*; `find -1 -d '^From ' HardToSpell` will find all occurrences of *HardToSpell* with one spelling error, and will output all mail messages containing any such occurrence (mail messages are separated by the pattern *From* with starts at the beginning of a line); `find -B Schwarzkopf` will give all occurrences of the *best match* to Schwarzkopf.

3.3. Preliminary Experiments

We experimented with three types of data, the personal file system of the first author, the Telecom archives, maintained at MIT, containing information related to telecommunications, and the library card catalog of the University of Arizona. Even though PIRS was originally designed for the first type of data, it worked very well for the second one and moderately well for the third one. None of the experiments was comprehensive and they are presented here just to get a very rough idea of the performance of PIRS. In particular, the timings depend on the load of the system as well as other factors, and they varied sometimes up to 40% on repeated tries. Since we expect to improve PIRS quite a bit in the near future, there is no reason to test the current version thoroughly.

The file system we indexed contained 36MB of text in 3538 different files. The index size was 1.01MB, which is 2.8% of the total. A typical search takes from 2-20 seconds. For example, a search for *Biometrics* took 2 seconds (0.2 seconds of user time); there was only one match. (These and all other numbers listed here were obtained through the UNIX time facility.) A search for *Scandinavia* found 17 matches in 13 seconds. A search for *fingerprint* took 25 seconds because there were 111 matches. These three searches took virtually the same amount of time when one spelling error was allowed! A Boolean search can be slower, but again it depends on the number of matches. The search for "protein AND matching" took 24 seconds yielding 6 matches; the search for "Griswold AND Andrews" took 16 seconds, finding again 6 matches; A

search for "information retrieval AND WAIS" took 28 seconds finding 2 matches.[3]

The Telecom archives contain about 100 files and 70.2MB. Because the number of files is quite small and they are already of the right size the index construction was very efficient and took less than 4 minutes. The index size was 1.46MB, which is 2% of the total. A search for *Schwarzkopf* allowing 2 errors (which were needed) took 3 seconds (only one match). A search for *MultiQuest* took 6 seconds finding matches in 4 different blocks. A search for *Sacramento* allowing one error took 49 seconds, because there were 168 matches in 64 blocks (more than half the blocks). That search, however, found 3 misspellings. If the information is already organized in files of the right size (about 1MB), then PIRS is much more efficient. Most searches were just as fast as in the previous example although the information base is twice as big.

The library catalog is a two-year old plain text version that was obtained from the library tapes. The size of the text file is 258MB describing approximately 2.5 million volumes. We divided the large file into 413 smaller files according to the first two characters of the call numbers. The index occupies 7.1MB, which is 2.8% of the size of the text. A search for an author, for example *Manber*, which yields one match, took 9 seconds of elapsed time. This is an example of an efficient query because there was only one match in one block. A search for *Salton* took 31 seconds of elapsed time yielding 17 matches (divided evenly between author names and books about the Salton sea[4]). A search for *UNIX* yielded 70 matches in 8 blocks, and it was still reasonably fast; it took 8.4 seconds of user time, 15.4 seconds of system time, and 41 seconds of elapsed time to output all matches. The search for *retrieval*, on the other hand, took 1:09 minutes of elapsed time due to the fact that 173 books on retrieval were found with 32 different call numbers prefixes. It was very interesting, however, to try to match retrieval allowing one spelling error. The search took 1:32 minutes of elapsed time, but it found nine additional books. The title of seven of them was truncated (the text we have stores titles in a fixed-size record of 40 characters) to show *retrieva*, and two others had *retrievable* in the title. A worse example was found by looking at the keyword *algorithm* allowing one error, and finding it misspelled *alogorithm* for one entry. With two errors, I also found *alogrithms*.

4. Future Plans

We list here briefly some avenues that we are currently exploring.

---

[3] This search requires that "information retrieval" appears together as one pattern and WAIS appears anywhere on the line. Three terms - information, retrieval, and WAIS - are searched in the index and then only two terms, "information retrieval", and "WAIS" are searched in the actual files. Such a combination would not be normally supported by an IR system.

[4] There is no inherent reason for us not to divide the text and index to categories such as author names, titles, and call numbers, allowing more structured queries. It just requires more processing work, and this is just a preliminary experiment.

4.1. Improving the query time

The running times reported in section 3.3 come from a prototype that can be greatly improved. One way to cut the query time is to divide the index into a list of words and one pointer (into another index that has the list of blocks) per word. This way the first-level search, which consists of a sequential search of the index file, will have a smaller file to search. We are also working on improving the performance of Boolean queries using a slightly larger index.

4.2. Incremental Indexes

The two-level index is easier to modify and adapt to a dynamic environment than a regular inverted index, because it is so much smaller. To add a new file to the current index, we first add the file to an existing block or create a new block if the file is large enough or important enough to deserve it. Then we scan the index and add each word in the new file that does not already appear in the block. Since the index is small, reading it from disk and writing it back can be done very fast. Deletion of a file is slightly more time consuming because for each word we need to check the whole block to determine whether that word appears somewhere else (and thus should not be deleted). Fortunately, agrep contains a very powerful algorithm for multi-pattern matching. It can search for a large collection of words (up to 30,000) concurrently. We will need to extend agrep, however, because it currently reports all the matches, while we need it to mark the words that had a match. But this is not too difficult to do.

Incremental indexing will be essential for indexing newsfeed. We are considering adapting PIRS to read usenet *netnews* messages by context. Currently, the total size of typical usenet server is from 500-800MB. Quite a bit of it is not text but images and programs, so it is definitely a size we can handle. The problem is that this kind of newsfeed consists of a large number of small files (individual email messages) stored at random places on the disk. A better data organization will probably be required.

4.3. More User Control

There are several ways to let the user improve the index by customizing. The user should be able to exclude files from the index by putting their name in a special file. The user can supply a list of stop words that should not be indexed; we currently index everything. The user should be able to decide whether certain patterns are indexed or not; for example, numbers, or words that include non-alphanumeric characters may or may not be indexed. Another option is to partition the collection of files into categories and build separate indexes for each (e.g., correspondence, information from servers, program source codes). There should be a way for the user to make those decisions in a convenient way, for example, by supplying a list of choices. We also plan to support special access to any special structure or additional information associated with the text. For example, some searches may want to specify that the desired information starts at column 30 on the line or in the second field. The user may want to specify that the search will include only small files (say, below 20KB), recent files (say, after August 1992), or files that contain only e-

mail messages. There should be more ways to control the output. For example, for queries that give many matches the user may be interested first in a rough idea of where those matches are (e.g., only directory names). (We currently have an option [-c] to list only the files containing a match along with the number of matches.) All these options can be incorporated in PIRS rather easily.

4.4. Text Compression

If the text is kept in a compressed form, it will have to be decompressed while the sequential search is performed. This will generally slow down the search considerably. But we are developing new text compression algorithms that may actually *speed up* the search. The first algorithm allows agrep (and most other sequential search algorithms) to search the compressed file directly without having to decompress it except when a match occurs and an output is generated, and then only a small neighborhood of the output is decompressed. (Essentially, instead of restoring the text back to its uncompressed form, we modify the pattern to fit the compressed form. The details are beyond the scope of this paper.) The search takes the same amount of time as a regular search in an uncompressed file of the same size, but the since the file is smaller than the original (being compressed), the search is faster. In preliminary tests, we achieved compression rates of about 30%, which are not competitive with good text compression methods, but are still useful for our purposes. In particular, we intend to compress the index used in the two-level scheme, because it is searched all the time. The second approach we are working on is designing a new text compression scheme with very fast decompression time (but slower compression times), so that decompression can be done on the fly during the search.

4.5. Interface to Other Systems

We are currently working on adapting the two-level approach and interfacing PIRS with distributed applications. For example, PIRS can be interfaced with WAIS servers [KM92], extend the *archie* servers [ED92], or be used in connection with the *Essence* program [HS93]. This work is just starting.

References

[BN91]

Bradford R., and T. Nartker, "Error correlation in contemporary OCR systems," *Proc. of the First Int. Conf. on Document Analysis and Recognition,* (1991), pp. 516–523.

[ED92]

Emtage E., and P. Deutsch, "Archie — An electronic directory service for the Internet," *Proc. of the USENIX Winter Conference,* San Francisco (January 1992), pp. 93–110.

[Fa85]
Faloutsos C., "Access methods for text," *ACM Computing Surveys*, 17 (March 1985), pp. 49–74.

[GB91]
Gonnet, G. H. and R. A. Baeza-Yates, *Handbook of Algorithms and Data Structures* (Chap. 7.2.6) Second Edition, Addison-Wesley, Reading, MA, 1991.

[HS93]
Hardy D. R., and M. F. Schwartz, "Essence: A resource discovery system based on semantic file indexing," to appear in *Proc. of the USENIX Winter Conference*, San Diego (January 1993).

[JD88]
Jain, A. K., and R. C. Dubes, *Algorithms for Clustering Data*, Prentice Hall, Englewood Cliffs, 1988.

[KM92]
Kahle B., and A. Medlar, "An information system for corporate users: Wide area information servers," *ConneXions — The Interoperability Report*, 5 (11) (November 1991), Interop Inc., pp. 2–9.

[MS93]
Manber U., and J. Small, "Using approximate matching in a pathology lab to handle misspellings of names," in preparation.

[RKN92]
Rice, S. V., J. Kanai, and T. A. Nartker, "A report on the accuracy of OCR devices," Technical Report, Information Science Research Institute, University of Nevada, Las Vegas, 1992.

[SM83]
Salton G., and M. J. McGill, *Introduction to Modern Information Retrieval*, McGraw-Hill, New York, 1983.

[WM92]
Wu S., and U. Manber, "Fast Text Searching Allowing Errors," *Communications of the ACM* 35 (October 1992), pp. 83–91.