Large Text Searching Allowing Errors

Márcio Drumond Araújo[1] Gonzalo Navarro[2 3] Nivio Ziviani[1 3]

[1] Depto. de Ciência da Computação, Universidade Federal de Minas Gerais, Brazil.
[2] Depto. de Ciencias de la Computación, Universidad de Chile, Chile.
[3] This work has been supported by Brazilian CNPQ Project 520916/94-8, Project RITOS/CYTED and Chilean Fondecyt grants 1960881 and 1950622.
E-mail: drumond@dcc.ufmg.br, gnavarro@dcc.uchile.cl, nivio@dcc.ufmg.br

Abstract. We present a full inverted index for exact and approximate string matching in large texts. The index is composed of a table containing the vocabulary of words of the text and a list of positions in the text corresponding to each word. The size of the table of words is usually much less than 1% of the text size and hence can be kept in main memory, where most query processing takes place. The text, on the other hand, is not accessed at all. The algorithm permits a large number of variations of the exact and approximate string search problem, such as phrases, string matching with sets of characters (range and arbitrary set of characters, complements, wild cards), approximate search with nonuniform costs and arbitrary regular expressions. The whole index can be built in linear time, in a single sequential pass over the text, takes near 1/3 the space of the text, and retrieval times are near $O(\sqrt{n})$ for typical cases. Experimental results show that the algorithm works well in practice: for a one-gigabyte text collection, all matchings of a phrase of 3 words allowing up to 1 error can be found in approximately 6 seconds and allowing no errors can be found in under half a second. This index has been implemented in a software package called *Igrep*, which is publicly available. Experiments show that Igrep is much faster than Glimpse in typical queries.

1 Introduction

The full text model in information retrieval (IR) is gaining popularity. In this model, documents are represented by their complete full texts. The user expresses his information needs by providing strings to be matched and the information system retrieves those documents containing the user specified strings. When the text collection is large it demands specialized index techniques for efficient text retrieval. A simple and popular indexing technique is the inverted list. It is especially adequate when the pattern to be searched for is formed by simple words. This is a common type of query, for instance when searching the World Wide Web, and therefore inverted lists have been widely used in that context.

One weakness of commercially available large text searching systems is the need for exact spelling due to the use of hashing or tree structures in the index. However, in many situations the pattern and/or the text are not exact, due to optical character recognition, typing or misspelling errors or because we are looking for approximate patterns. For example, a name we are looking for may be misspelled in the text or we may not remember its exact spelling. The approximate text searching problem is to find all substrings in a text database that are at a given "distance" k or less from a pattern p. The *distance* between two strings is the minimum number of insertions, deletions or substitutions of single characters in the strings that are needed to make them equal. The case $k = 0$ corresponds to the classical exact matching problem.

The classical solution for approximate searching is $O(mn)$ time, where m is the size of the pattern and n is the size of the text [Sel80]. Since the beginning of the eighties there is a long list of papers on the subject, where [BYG92, WM92, CL92, ST95, BYP92, WMM96, BYN96a, Nav97] is a partial list of the most recent ones.

From the practical point of view an important new paradigm called *bit-parallelism* was developed by Baeza-Yates and Gonnet [BYG92]. In their algorithm the state of the search is represented as a number and only bitwise logical operations shifts and additions are used. Wu and Manber [WM92] extended this numeric scheme to deal with the more general approximate string matching problem under some editing distance. They present a $O(kn)$ algorithm (where k is the number of errors) that supports a large number of variations of the problem. Recently, this algorithm has been improved to $O(n)$ for small patterns (e.g. up to 9 letters on a 32-bit architecture) [BYN96a].

On the other hand, the problem of finding good indexing schemes that allow approximate searching was considered in [WM92, BY92] the unresolved problem in this area. There are many different linear time approximate string matching algorithms, but only recently there is some work done for the case when the text is large and an index must be built to speed up the search.

We can distinguish two different indexing models. The first is capable of retrieving *any substring* of the text whose edit distance to the pattern is sufficiently small. The second retrieves only *complete words*, whose edit distance to the pattern is small enough. For instance, only the first model will find `"shallow"` with one error in the text `"...sha llow..."`, although it will also find that pattern in the text `"...hash allows..."`, what we probably don't want. Although the first model is more general, the second one may be better suited for IR purposes on natural language text. Moreover, most indices for the first model are still in a preliminary stage: indices are too large and no disk storage strategies have been devised yet. The implementations are in general very primitive prototypes. Examples of these indices are [Ukk93, Cob95, BYNST96, ST96, LST96, Mye94].

This work focuses on word-retrieving indices. One successful attempt to solve this problem was presented by Manber and Wu [MW93] in a system called Glimpse. They propose a two-level information retrieval structure that combines a partial inverted file with sequential searching. They divide the text into nearly 256 blocks of the same size and build an index of all different words plus a list of the blocks where each word appears. Approximate queries are handled by first using an on-line algorithm (Agrep [WM92]) on the vocabulary to find all words in the index that match approximately with the pattern, and then the corresponding blocks are searched, using Agrep again, to find the particular matches. In the worst case, it may be necessary to search all the blocks, which makes Glimpse adequate for use with intermediate large text collections (say up to 200 megabytes). Baeza-Yates and Navarro [BYN97] study an alternative scheme where the text is not searched for the approximate pattern but with a multipattern search of all the words in the vocabulary that matched the pattern. They also prove that it is possible to have an index which is sublinear in space and time simultaneously and study the practical effect of the block size.

In this paper we present an efficient word-retrieving indexing scheme for large text searching, which is fast at indexing and querying time and has the capability of searching exactly or allowing errors in the pattern and/or in the text. The index can be built in $O(n)$ time and takes $O(n)$ space. Querying performance is near $O(\sqrt{n})$ time. The implementation of the algorithm has been tested successfully for files with more than 1 gigabyte of text. It supports a large number of variations of the approximate string search problem. In addition to single words and phrases, the system supports string matching with sets of characters (range and arbitrary set of characters, complements, wild cards), nonuniform costs and arbitrary regular expressions.

The algorithms presented in this paper are being used in a software package called *Igrep*. Igrep is an approximate matching tool for very large text collections. The software package is a prototype in its version 1.0, which is available from `ftp://dcc.ufmg.br/pub/research/~nivio/-igrep`.

2 Structure of the Index

We present an index based on the traditional inverted list model. We view a text file as a sequence of *words*, separated by the usual delimiters (e.g. space, end-of-line, period, comma). We scan the whole text, word by word, build a table containing all different words (the *vocabulary*) of the text and store every occurrence of each word on a list. The list of occurrences of each word are kept in order by position in the text. Figure 1 illustrates the structure of the index for an example of a text database with six words. Each entry of the table of words contains a word and a pointer to the end of its list of occurrences. A brief description of the index construction can be found in Section 3.

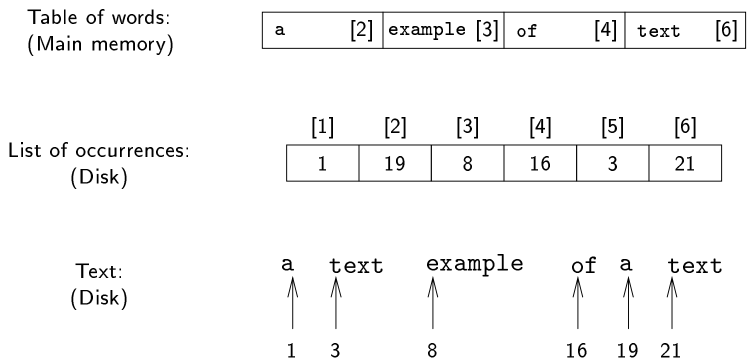

Fig. 1. Structure of the index

To answer a query the searching procedure needs only the table of words and the list of occurrences, making the text itself not necessary at all. The size of the vocabulary of any large literary text is very small if compared to the size of the text, and so the table of words can be kept in main memory all the time (more about the size of the vocabulary can be found in Sections 3.2 and 5). For a single word pattern we just perform a search in the table of words for the list of occurrences that contains all the matches of the pattern. When the pattern is more than one word long (phrase pattern) we first search the table for each word of the pattern and retrieve the corresponding lists of occurrences. Next, we obtain the intersection of the lists looking for pointers that have the same relative positions they share in the pattern, thus obtaining the final answer.

To illustrate the searching procedure we present two examples. Exact searching for the pattern `text` in Figure 1 involves binary searching the table of words for the list interval $\langle 5, 6 \rangle$. To search for the pattern `text` `sample` with editing distance $k = 2$ in Figure 1 we search with $k \leq 2$ errors the first word `text` of the pattern and obtain one list interval $\langle 5, 6\rangle$ for $k = 0$. Next, we search with $k \leq 2$ errors the second word `sample` of the pattern and obtain the word `example` corresponding to the list interval $\langle 3, 3\rangle$ for $k = 2$. Now we end up with the two lists $\{8\}$ and $\{3, 21\}$ corresponding to the list intervals $\langle 3, 3\rangle$ and $\langle 5, 6\rangle$. The final answer is the list $\{3\}$, result of the intersection of the two lists, given that `text` and `sample` are at the proper distance in the pattern. In general we consider all lists related to each word of the query such that the total sum is $\leq k$. In the pattern `text sample` we had one list related to the first word `text` with $k = 0$ and one list related to the word `sample` with $k = 2$.

3 Index Construction

The procedure to build the index works as follows. We scan the text, word by word, find each word in a hash table and insert its text position at *the end* of the corresponding occurrence list. If a word is not present in the hash table, it is inserted and the corresponding occurrence list is initialized. The index is stored on disk in the format shown in Figure 1.

However, the resulting index most probably will not fit in main memory. When the main memory is exhausted, we store the partial index as if it were the complete final index. This partial index is called a *dump*. We then continue the process starting from scratch with a new dump.

Once we complete this process, we merge the dumps. Merging two dumps involves concatenating the lists of occurrences of each word, which takes linear time. Partial dumps are merged until the complete index is obtained. We can merge r dumps in a single process, in a fashion very similar to r-way list merging, at $O(n \log_2 r)$ time (e.g. using a heap). We tested different values of r and, although larger values produce better times in a reasonable range, the overall differences are too small to take into account. We perform an in place merging as described in [MB95].

3.1 Time to Build the Index

The cost to search a word in the hash table is $O(1)$ on average. As the text addresses always increase as the scanning goes on, the insertions in the list of occurrences happen always at the end of the lists, at $O(1)$ cost. Thus, the total CPU cost to build the dumps is $O(n)$ on average.

By keeping the words of the vocabulary in a trie instead of a hash table, the time cost can be made $O(n)$ in the worst case. This is because, for each one of the $O(n)$ *characters* of the text, we work $O(1)$ in the trie.

We analyze now merging times. Let M be the amount of main memory available. Given that we can merge groups of r dumps in a single process, we can divide the n/M dumps in groups of r, merge each group and obtain $n/(Mr)$ groups of larger dumps. This process is repeated until we have only one final index, as shown in Figure 2.

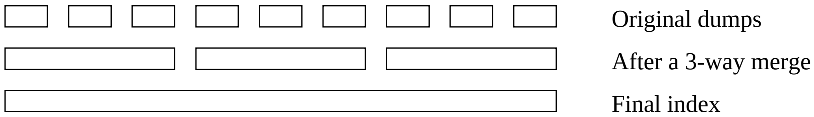

Fig. 2. The process of merging three dumps each time

Since the time to merge r groups of size M each is $O(Mr \log_2 r)$, the total amount of work in the first level is $O(n \log_2 r)$, which is the same for each iteration. Since there are $\log_r(n/M)$ iterations, the total amount of time is $O(n \log_2(n/M))$, which is independent of r. The value of r affects disk times, although the effect is barely noticeable.

Therefore the algorithm is $O(n \log n)$ on average. However, it can be made $O(n)$ in the worst case. If instead of dumping and merging we keep a separate file for each word in the vocabulary, for each word in the text we must add an occurrence to the end of its file, at $O(n)$ total cost. However, except for huge texts, dumping and merging is more practical because it avoids random accesses to disk. The algorithm could decide which strategy to employ based on the text size, this way keeping $O(n)$ all the time as well as choosing the fastest strategy for each case.

3.2 Space for the Index

It is empirically known that the vocabulary of a text with n words grows sublinearly. Moreover, the following relation holds very accurately [Hea78]

$$V = Kn^\beta = O(n^\beta) \tag{1}$$

where V is the size of the vocabulary and $0 < \beta < 1$ is a constant dependent on the particular text. We show later an experimental verification of this fact.

Hence, the larger part of the index is the list of occurrences, which is $O(n)$. Stop words represent approximately 30-40% of the text (see Section 5.2 for the definition of stop words). For each non-stop word, we store a pointer (4 bytes is enough in most cases), while the length of non-stop words is approximately 6-7 characters. This fact (that we later verify experimentally) yields $0.35n$, i.e. a 35% overhead over the text.

4 Querying

In our system there are basically two types of patterns: one word patterns and phrase patterns. In each case we can look for exact and approximate occurrences of the pattern in the text. Each of these four combinations involves quite different algorithms and tasks to be performed. Next, we describe the most representative combinations derived from the two basic types of patterns.

4.1 One Word Patterns

The most important characteristic of one word patterns is that only the vocabulary is consulted and the list or lists of occurrences are immediately retrieved. For example, simply searching a word retrieves its list of occurrences, searching for a word allowing errors or for a regular expression may retrieve more than one list as more than one word of the vocabulary may match the query.

Searching on the vocabulary can be binary or sequential. Exact searching a word involves a binary search on the vocabulary. Searching a regular expression or approximate searching of a word involves a sequential search on the vocabulary. For simple patterns allowing k errors we use the algorithm [BYN96a], which is $O(n)$ for small patterns, and extremely fast in practice. The algorithm is based on an automaton whose behavior is simulated in $O(1)$ per inspected character for short patterns. In a 32-bit architecture, words of length up to 9 can be searched in $O(n)$ with any number of errors, and up to length 11 with one error. This is good for our purposes, because most words are not longer than 9 letters in practice. Our experiments show that approximate searching on the vocabulary always takes less than a second with this algorithm.

In [BYN96a] a number of techniques are developed to cope with longer patterns. However, we take a different approach here. Since the few words longer than 9 letters will have a few more characters, we truncate them to the first 9 characters and use the algorithm as a filter. Each occurrence reported by the filter is checked with dynamic programming to verify if it involves a real match of the complete word.

As it is shown in [BYN96a], the number of verifications is extremely low if the error ratio is reasonably small. It is also shown that there is an abrupt division in the domain of error ratios. There is a point such that any query allowing more than that error ratio will retrieve a huge amount of information. Since this is of no use in terms of information retrieval (because of lack of precision) we focus only in the case of lower error ratios. It is possible to estimate beforehand the size of the result (to give the user early feedback on the precision of his query) at very low cost.

This automaton can have not only single letters in the pattern, but any set of characters at each position. This allows our system to support very efficiently the following extended queries (exactly or allowing errors):

- range of characters (e.g. `t[a-z]xt`, where `[a-z]` means any letter between `a` and `z`);
- arbitrary sets of characters (e.g. `t[aei]xt` meaning the words `taxt`, `text` and `tixt`);
- complements (e.g. `t[~ab]xt`, where `~ab` means any single character except `a` or `b`; `t[~a-d]xt`, where `~a-d` means any single character except `a`, `b`, `c` or `d`);
- arbitrary characters (e.g. `t·xt` means any character as the second character of the word);
- case insensitive patterns (e.g. `Text` and `text` are considered as the same words).

For more complicated patterns, allowing k errors or not, we use the algorithm [WM92], which is $O(kn)$ (and $O(n)$ with no errors). Processing the vocabulary with this algorithm takes typically 1-4 seconds. In addition to single strings of arbitrary size and classes of characters described above the system supports patterns combining exact matching of some of their parts and approximate matching of other parts, unbounded number of wild cards, arbitrary regular expressions, and combinations, as follows:

- unions (e.g. `t(e|ai)xt` means the words `text` and `taixt`; the expression `t(e|ai)*xt` means the words beginning with `t` followed by `e` or `ai` zero or more times followed by `xt`). In this case the word is seen as a regular expression;
- arbitrary number of repetitions (e.g. `t(ab)*xt` means that `ab` will be considered zero or more times). In this case the word is seen as a regular expression;
- arbitrary number of characters in the middle of the pattern (e.g. `t#xt`, where `#` means any character considered zero or more times). Note that `#` is equivalent to $\cdot *$ (e.g. `t#xt` and `t·*xt` obtain the same matchings but the latter is considered as a regular expression). In this case the word is not considered as a regular expression for efficiency because the treatment of a regular expression generally demands more bitwise operations than the `#` case;

- combining exact matching of some of their parts and approximate matching of other parts (e.g. `<te>xt`, with $k = 1$, meaning exact occurrence of `te` followed by any occurrence of `xt` with 1 error);
- matching with nonuniform costs (e.g. the cost of insertions can be defined to be twice the cost of deletions).

4.2 Phrase Patterns

For patterns containing more than one word we search each word separately on the vocabulary and then intersect the lists of occurrences. Each word of the phrase can be a simple word or a complex regular expression, and can allow errors as in Section 4.1. Exact searching a phrase involves searching each word on the vocabulary and intersecting the lists of occurrences. The final answer contains the intersection of the lists represented by that positions in the text with the same relative positions presented by the words in the pattern.

It is also possible to search a phrase allowing k errors in the whole phrase. This involves sequential searching of each word on the vocabulary with k errors and intersecting the lists of occurrences, taking care of the total number of errors. We keep a list of matches for each word and each number of errors and intersect each combination that has less than or equal to k total number of errors. For each word of the pattern a different algorithm is chosen, according to the many possibilities described in the previous section.

The intersection of many lists is carried out as follows: the shortest list is selected as a first version of the result. Then, it is intersected with each other list by binary searching the elements of the shorter lists inside the other (taking care of the positions of the words in the text). This works well because, as shown in the Appendix, it is very probable that one of the lists is very short.

4.3 Time to Answer a Query

In the Appendix we analyze each type of query. We use α as a shorthand for $1 - \beta$, and observe that $0 < \alpha < 1$. In natural language text β is between 0.4 and 0.6, hence $\alpha \approx \beta$ (see Section 5). The results are approximate (since the text models are only approximations) and valid for queries that have a reasonable degree of precision (i.e. queries useful to the user). As explained in the Appendix, γ is related to the number of allowed errors and is typically in the range 0.1 to 0.2.

- Simple words: $O(\log n)$.

– Phrases of simple words: $o(n^\beta)$ for two words, $O(\log n)$ for longer phrases.
– Extended patterns, regular expressions and approximate words: $O(n^\beta + n^{\alpha+\gamma} \log n)$.
– Phrases of the above patterns: $O(n^\beta + n^{\alpha+\gamma} \log n)$.
– Approximate phrase matching: $O(n^\beta + n^{\alpha+\gamma} \log n)$.

Therefore, except for some types of exact searches, retrieval times are in the range $O(n^{0.4..0.8})$ depending on the vocabulary size and the complexity of the search. In reasonable cases it is $O(n^{0.6})$, which is near $O(\sqrt{n})$. We also point out that the disk accesses to the index are sequential (except for buffering limitations).

5 Experimental Results

For the experimental results we used literary texts from the 2 gigabytes TREC collection [Har95]. We have chosen the following texts: AP Newswire (1989), DOE - Short abstracts from DOE publications, FR - Federal Register (1989), WSJ - Wall Street Journal (1987, 1988, 1989) and ZIFF - articles from *Computer Selected* disks (Ziff-Davis Publishing). We also derived two other larger files by putting together AP plus ZIFF texts (called AZ text file) and AP plus DOE plus FR plus WSJ plus ZIFF texts (called ADFWZ text file). Our objective here is to obtain two large files containing 458.2 megabytes and 1.09 gigabytes, respectively. Table 1 presents some statistics about the seven text files. For the WSJ file the vocabulary size (in bytes) is 0.58% of the text size and the number of words of the vocabulary is 0.49% of the total number of words. For our experiments we considered a word as a contiguous string of characters in the set {A..Z, a..z} separated by other characters not in the set {A..Z, a..z}.

The performance evaluation of the algorithms presented in the previous sections was obtained by means of 500 trials to query different text files and 20 repetitions to build indices. This gives a confidence interval of 95% for our measures.

The experiments show that our index is very efficient even for very large text files. All tests were run on a SUN SparcStation 4 with 128 megabytes of RAM running Solaris 2.5.1.

Files	Text		Vocabulary		Vocab./Text	
	Size (bytes)	Words	Size (bytes)	Words	Size	Words
AP	237,766,005	37,740,089	1,530,192	201,115	0.64%	0.53%
DOE	180,515,212	27,124,239	1,795,783	211,196	0.99%	0.78%
FR	219,987,476	32,000,223	1,043,869	132,129	0.47%	0.41%
WSJ	262,757,554	40,741,508	1,511,951	198,818	0.58%	0.49%
ZIFF	242,660,178	38,047,824	1,639,677	216,482	0.68%	0.57%
AZ	480,426,183	75,787,913	2,574,518	336,716	0.54%	0.44%
ADFWZ	1,143,686,425	175,653,883	4,629,371	573,661	0.40%	0.32%

Table 1. Text files from the TREC collection

5.1 Time to Build the Index

Table 2 presents the times to build the index for three different files containing 250.6, 458.2 and 1090.7 megabytes of text, respectively. The third column shows the time devoted to merging times. In this case the times were obtained for a 2-way merge (i.e. $r = 2$).

As can be seen, the indexing times are almost linear with the size of the text. In our machine, indexing performance is near 4 megabytes per minute.

File	Size (megabytes)	Total time (min)	Merge time (min)	Mb/min
WSJ	250.6	58.5	13.8	4.28
AZ	458.2	122.7	33.9	3.73
ADFWZ	1090.7	248.9	79.8	4.38

Table 2. Experimental results to build the index

5.2 Space for the Index

Table 3 presents the worst case and average case (n/V) for the sizes of the lists of occurrences for the texts AP, DOE, FR, WSJ, ZIFF, AZ and ADFWZ. Note that in all seven texts the largest size for the list of occurrences corresponds to the word `the`.

The majority of the most common words in natural languages are function words (also called stop words) whose purpose is mainly syntactical and do not carry enough content to occur alone in the query. An interesting study of English texts by Miller, Newman and Friedman [MNF58] classifies the words into function words (articles, prepositions, pronouns, numbers, conjunctions and auxiliary verbs) and content words (nouns, verbs, adjectives and most adverbs).

Files	Words Text(n)	Words Voc.(V)	Most freq. word		$\frac{n}{V}$	Index size (bytes)	$\frac{\text{Index}}{\text{Text}}$
			Word	Occ.			
AP	37,740,089	201,115	the	2,077,987	188	152,490,548	0.64
DOE	27,124,239	211,196	the	1,722,275	128	110,292,739	0.61
FR	32,000,223	132,129	the	2,066,443	242	129,044,761	0.59
WSJ	40,741,508	198,818	the	2,020,113	205	165,989,934	0.63
ZIFF	38,047,824	216,482	the	1,556,762	176	153,830,973	0.63
AZ	75,787,913	336,716	the	3,634,749	225	305,726,170	0.64
ADFWZ	175,653,883	573,661	the	9,443,580	306	707,244,903	0.62

Table 3. Size of the lists of occurrences, including stopwords

Table 4 presents the influence of a set of 361 function words obtained from [MNF58] in the five files. For the WSJ file, the 361 words, which are less than 0.18% of the vocabulary of 198,818 words, account for 44% of all 40,741,508 word occurrences (our software is case sensitive so we considered each stop word twice, starting with lower case and upper case letters). By eliminating function words the worst and average lengths of the lists of occurrences are much closer to what actually happens in practice, as we always try to use content words when retrieving information from text databases. Moreover, our index takes approximately 35% of the space of the text when the stop words are not indexed, which is the option in general for information retrieval systems.

Files	Words Text(n)	Words Voc.(V)	Most freq. word		$\frac{n}{V}$	Index size (bytes)	$\frac{\text{Index}}{\text{Text}}$
			Word	Occ.			
AP	20,678,146	200,392	said	504,998	103	84,239,271	0.35
DOE	15,515,153	210,523	energy	61,748	73.7	63,853,146	0.35
FR	17,526,092	131,457	Section	104,490	133	71,145,018	0.32
WSJ	22,833,202	198,079	said	303,618	106	92,841,139	0.35
ZIFF	21,197,303	215,753	software	110,723	98.2	86,425,334	0.36
AZ	41,875,449	335,974	said	576,987	125	170,072,682	0.35
ADFWZ	97,749,896	572,903	said	885,374	171	395,625,209	0.35

Table 4. Size of the lists of occurrences, excluding stopwords

5.3 Time to Answer a Query

The experiments to measure query times considered exact and approximate queries ($k = 0, 1, 2, 3$), phrase patterns containing 1, 2, 3, 4, and 5 words and the texts WSJ, AZ and ADFWZ. The patterns were randomly chosen from the texts, but avoiding patterns containing function words.

We tested our software against Glimpse version 3.0 [MW93] for the WSJ file, using the same set of queries used for our software package Igrep. For this experiment we used the option `-b`, when Glimpse builds an index 16.9% of the size of the text (index size of 42.4 megabytes), allowing faster search. This option forces Glimpse to store an exact pointer to each occurrence of each word (i.e. a full inverted index), except for some very common words belonging to a stop list it always uses in this case. Results are shown in Table 5.

	1 word		2 words		3 words		4 words		5 words	
k	t	r	t	r	t	r	t	r	t	r
0	0.08	0.3%	0.23	0.9%	0.24	1%	0.28	1%	0.34	1%
1	0.58	0.4%	1.99	1.5%	2.15	1.6%	2.59	1.9%	3.16	*
2	0.85	0.5%	8.27	5.1%	4.26	2.6%	4.65	2.9%	5.06	*
3	1.30	0.7%	34.1	17.9%	14.6	7.5%	11.2	*	8.97	*

* Glimpse does not accept queries allowing errors with more than 32 characters

Table 5. Igrep searching times in seconds (t) and ratio Igrep/Glimpse (r) for the WSJ text

Tables 6 and 7 show the results using Igrep for the larger files AZ (458.2 megabytes) and ADFWZ (1090.7 megabytes), respectively. We did not run Glimpse for these two files because its query times are too long on very large texts. Our approach, instead, works well with texts of 1 gigabyte and more.

	Igrep				
k	1 word	2 words	3 words	4 words	5 words
0	0.087 ± 0.004	0.32 ± 0.03	0.33 ± 0.02	0.35 ± 0.02	0.40 ± 0.02
1	0.95 ± 0.01	3.3 ± 0.3	3.8 ± 0.1	4.3 ± 0.1	5.2 ± 0.1
2	1.4 ± 0.1	13 ± 3	7.0 ± 1.0	7.2 ± 0.5	8.2 ± 0.4
3	2.2 ± 0.1	70 ± 21	7.1 ± 0.6	12 ± 2	14 ± 2

Table 6. Searching times (in seconds) for the AZ file text using Igrep

The only case in which our index does not work well is for phrases of two words searched with 3 errors or more. This agrees with the analysis, in the sense that two words are not enough to guarantee that one of them has a sufficiently small list of occurrences. Three errors imply searching both words with three errors, and later intersecting the appropriate lists. A word searched with three errors generates a huge list of matches in the vocabulary.

Igrep					
k	1 word	2 words	3 words	4 words	5 words
0	0.095 $\pm$ 0.006	0.46 $\pm$ 0.05	0.41 $\pm$ 0.03	0.44 $\pm$ 0.03	0.48 $\pm$ 0.03
1	1.6 $\pm$ 0.1	6.2 $\pm$ 0.6	6.1 $\pm$ 0.3	7.1 $\pm$ 0.2	8.4 $\pm$ 0.2
2	2.3 $\pm$ 0.1	38 $\pm$ 10	18 $\pm$ 3	15 $\pm$ 2	15 $\pm$ 1
3	3.5 $\pm$ 0.1	108 $\pm$ 30	52 $\pm$ 12	34 $\pm$ 8	37 $\pm$ 11

Table 7. Searching times (in seconds) for the 1 gigabyte ADFWZ text file using Igrep

A possible solution is to forbid more than 2 errors in a single word of a phrase. Another one involves using the text at query time: instead of generating all the matches of a word with 3 errors, generate those of the other one with zero errors and check directly in the text whether the whole phrase appears with 3 errors. Thus, the huge list of matches is never generated.

The following test was for more complicated patterns, as follows:

1. <exe>cutive: meaning exact occurrence of exe followed by any occurrence of cutive with k errors.
2. prob#atic sign#ance: where # means any character considered zero or more times (one possible answer is problematic significance).
3. <[LMN]ACM># received: meaning a word starting by L, M, or N followed by ACM followed by any character considered zero or more times followed by the word received (one possible answer is LACM received). For this example, the search is case insensitive for both Igrep and Glimpse.
4. earl# retir[aeiou]#<ent> program: the #, [] and <> meaning as before (one possible answer is early retirement program).
5. acc[aeiou]*unt compri[ms](es|ent): pattern is a regular expression (one possible answer is account comprises).

Table 8 presents searching times and ratio against Glimpse using the WSJ file for $k = 0, 1, 2, 3$, for the five patterns above.

Table 9 presents experimental results for the values of K and β from Eq.(1) and θ from Eq.(2). From the values obtained for β we can conclude that retrieval times are near $O(\sqrt{n})$ for typical texts.

6 Conclusions and Future Extensions

We have presented an indexing scheme capable of retrieving words and phrases, exact and approximate search, using classes of characters and

Pattern	$k=0$		$k=1$		$k=2$		$k=3$	
	t	R	t	R	t	R	t	R
1	0.031	0.004%	3.18	0.028	6.92	0.047	11.4	*
2	3.64	0.048	12.4	0.1	20.7	0.126	30.1	0.171
3	3.74	0.047	11.7	0.1	19.3	0.133	29.0	0.169
4	4.60	0.061	14.5	0.124	38.0	0.26	191	*
5	10.2	8.4%	22.2	8.8%	**	**	**	**

* For Glimpse k must be smaller than the number of characters between < and >
** The number of errors must be smaller than the number of characters of the smallest sequence between (and) in a regular expression

Table 8. Igrep searching times in seconds (t) and ratio against Glimpse (R) for the WSJ text

Text	AP	DOE	FR	WSJ	ZIFF	AZ	ADFWZ
K	26.8	10.8	13.2	43.5	11.3	9.2	4.8
β	0.46	0.52	0.48	0.43	0.51	0.52	0.56
θ	1.87	1.70	1.94	1.87	1.79	1.85	1.85

Table 9. Experimental results for the coefficients of Heaps and Zipf equations

general regular expressions. It is based on full inverted lists, where the work is done on the vocabulary and the text is not accessed at all. This allows to work with texts stored on remote, slow or removable devices, or even with no text at all. The index is implemented as a software package called Igrep, which is publicly available.

Our analytical and experimental results show that the performance of the index is good even for text files of more than 1 gigabyte. The index can be built in linear time and a single pass over the text (in our machine the average indexing speed is 4 megabytes per minute), and takes linear space (35% of the text size is typical). Querying performance is near $O(\sqrt{n})$ for queries that are useful in terms of precision. Typical times for one gigabyte of text are a few seconds for useful queries.

We are currently working on extensions of this index. It is easy to extend the index to handle collections instead of single files, and restrict the queries to some subcollections. Reindexing is also easy, since it is sufficient to index again the files that were added or updated and merge the original and the differential indices, which can be efficiently done and allows to use the original index until the last minute. This lightweight reindexing capability is very good in the Web environment, where changes are continuous but not extensive.

We are also studying the best way to handle approximate phrase searching, to compare the current approach to the one of verifying directly in the text.

Finally, we are working on integrating compression techniques, to make the whole index plus compressed text nearly half of the original text [MNZ97].

Acknowledgements

We wish to acknowledge the helpful comments of Ricardo Baeza-Yates.

References

[BY92] R. Baeza-Yates. Text retrieval: theory and practice. In *Proc. of 12th IFIP World Computer Congress*, volume I, pages 465–476, 1992. Elsevier Science.

[BYG92] R. Baeza-Yates, G.H. Gonnet. A new approach to text searching. *Communications of the ACM*, 35(10): 74–82, 1982.

[BYN96a] R. Baeza-Yates and G. Navarro. A faster algorithm for approximate string searching. In *Proc. CPM'96*, Springer-Verlag LNCS, v. 1075, pages 1–13, 1996.

[BYN97] R. Baeza-Yates and G. Navarro. Block addressing indices for approximate text retrieval. Tech. Report TR/DCC-97-3, Dept. of CS, Univ. of Chile. Submitted.

[BYNST96] R. Baeza-Yates, G. Navarro, E. Sutinen and J. Tarhio. Indexing methods for approximate text retrieval. Tech. Report TR/DCC-97-2, Dept. of CS, Univ. of Chile.

[BYP92] R. Baeza-Yates and C. Perleberg. Fast and practical approximate pattern matching. In *Proc. CPM'92*, Springer-Verlag LNCS, v. 644, pages 185–192, 1992.

[CL92] W. Chang and J. Lampe. Theoretical and empirical comparisons of approximate string matching algorithms. In *Proc. CPM'92*, Springer-Verlag LNCS, v. 644, pages 172–181, 1992.

[Cob95] A. L. Cobbs. Fast approximate matching using suffix trees. In *Proc. CPM'95*, Springer-Verlag LNCS v. 937, pages 41–54, 1995.

[GBY91] G. H. Gonnet and R. Baeza-Yates. Handbook of Algorithms and Data Structures. Addison-Wesley, 1991.

[Har95] D. K. Harman. Overview of the third text retrieval conference. In *Proc. Third Text Retrieval Conference (TREC-3)*, pages 1-19, NIST Special Publication 500-207, Gaithersburg, Maryland, 1995.

[Hea78] J. Heaps. *Information Retrieval - Computational and Theoretical Aspects.* Academic Press, NY, 1978.

[LST96] O. Lehtinen, E. Sutinen and J. Tarhio. Experiments on block indexing. In *Proc. Third South American Workshop on String Processing (WSP'96)*, Carleton University Press International Informatics Series, v. 4, pages 183–193, 1996.

[MB95] A. Moffat and T. Bell. In situ generation of compressed inverted files. *Journal of the American Society for Information Science* 46(7):537-550, 1995.

[MNZ97] E. de Moura, G. Navarro and N. Ziviani. Indexing compressed text. In R. Baeza-Yates, editor, *Proceedings Fourth South American Workshop on String Processing*, Carleton University Press International Informatics Series, Valparaiso, Chile, 1997.

[MNF58] G. A. Miller, E. B. Newman and E. A. Friedman. Length-frequency statistics for written English. *Information and Control* 1:370-380, 1958.

[MW93] U. Manber and S. Wu. GLIMPSE: A tool to search through entire file systems. Tech. Report 93-34, Dept. of CS, Univ. of Arizona, Oct 1993.

[Mye94] E. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica* 12(4/5):345–374, 1994. Springer-Verlag.

[Nav97] G. Navarro. Approximate string matching by counting. Tech. Report TR/DCC-97-1, Dept. of CS, Univ. of Chile. Submitted.

[Sel80] P. Sellers. The theory and computation of evolutionary distances: pattern recognition. *J. of Algorithms*, 1:359–373, 1980.

[ST95] E. Sutinen and J. Tarhio. On using q-gram locations in approximate string matching. In P. Spirakis, editor, *Proc. ESA'95*, Springer-Verlag LNCS v. 979, Corfu, Greece, pages 327–340, 1995.

[ST96] E. Sutinen and J. Tarhio. Filtration with q-samples in approximate string matching. In *Proc. CPM'96*, Springer-Verlag LNCS v. 1075, pages 50–61, 1996.

[Ukk85] E. Ukkonen. Finding approximate patterns in strings. *Journal of Algorithms* 6:132-137, 1985.

[Ukk93] E. Ukkonen. Approximate string-matching over suffix trees. In *Proc. CPM'93*, Springer-Verlag LNCS 684, pages 228–242, 1993.

[WM92] S. Wu, U. Manber. Fast text searching allowing errors. *Communications of the ACM*, 35(10):83–91.

[WMM96] S. Wu, U. Manber, and E. Myers. A sub-quadratic algorithm for approximate limited expression matching. *Algorithmica*, 15(1):50–67, 1996.

[Zipf49] G. Zipf. *Human Behaviour and the Principle of Least Effort.* Addison-Wesley, 1949.

Appendix: Times for Different Querying Operations

There are a number of different types of query to analyze. Each type involves carrying out different tasks. For example, simply searching a word involves a binary search on the vocabulary; searching a phrase involves binary searching each word and then intersecting the lists; searching for a regular expression involves a sequential search on the vocabulary plus merging the resulting lists.

We remark that this analysis is approximate, since it relies on rules such as the Heaps law or the Zipf law, which are only rough approximations to the statistical structure of texts. Moreover, the results are valid only for queries useful to the user (i.e. with reasonable degree of precision).

We first analyze the cost of each task, and use the results to deduce the cost of each type of query. The description of the tasks follow, together with their analysis. Recall that the size of the vocabulary is $V = O(n^\beta)$ and that $\alpha = 1 - \beta$, where α and β are normally in the range 0.4 to 0.6 [Hea78].

bin-search Binary searching a word in the vocabulary and retrieving the list. Since the search is binary, we have $O(\log n^\beta) = O(\log n)$ cost for this type of task.

seq-search Sequential searching a word in the vocabulary is $O(n^\beta)$. This is the case of regular expressions and others. It is also the case of approximate simple word matching, since, as explained, we use a linear-time filter and the number of verifications is not significant in practice. Searching a complex expression with k errors, on the other hand, is $O(kn^\beta)$. We take k as a constant.

lst-merge List merging of j lists occurs in approximate search, non-standard patterns, etc. Since the average size of each list of occurrences is $n/V = O(n^{1-\beta})$ and we merge ordered lists to produce an ordered list, we work $O(n^\alpha j \log j)$.

lst-inters List intersection of j lists occurs in phrases. Those lists can come from searching simple words or complex expressions. In the latter case we use an algorithm similar to that of list merging to achieve $O(j\ell \log j)$ for j lists of length ℓ. In the case of simple words we select the smaller list and, for each element, binary search adjacent positions in the other lists. We show that in this case the length of the shortest list is $O(1)$ on average, so we work $O(j \log n)$ on average. To show that, we assume valid the generalized Zipf law

[Zipf49, GBY91], which says that the number of occurrences of the i-th more frequent word is, for some θ dependent on the text,

$$f(i) = \frac{n}{i^\theta s(\theta)}, \qquad \text{where} \qquad s(\theta) = \sum_{i=1}^{V} \frac{1}{i^\theta} \tag{2}$$

which is constant for $\theta > 1$. We experimentally validated this law in Section 5.

It must hold $f(V) = O(1)$, i.e. there exist words that appear once (there are a lot, in fact). Under the model $V = O(n^\beta)$ we have $\theta = 1/\beta$ for sufficiently large texts (e.g. ADFWZ). If we consider $X_1..X_j$ the rank of the words present in a phrase (which are uniformly distributed over $[1..V]$), we have

$$\begin{aligned} P(\min f(X_i) \;\geq a) &= (P(f(X_1) \geq a))^j \\ &= \left(P\left(X_1 \leq \left(\frac{n}{a\ s(\theta)}\right)^{1/\theta}\right)\right)^j \\ &= \left(\left(\frac{n}{a\ s(\theta)}\right)^{1/\theta} \frac{1}{V}\right)^j = (a\ s(\theta))^{-j/\theta} \end{aligned}$$

Hence, the expectation of the shortest list is

$$\sum_{a=1}^{V} P(\min f(X_i) \;\geq a) \quad \leq \quad \frac{1}{s(\theta)^{j/\theta}} \sum_{a \geq 1} \frac{1}{a^{j/\theta}}$$

which is $O(1)$ for $j > \theta$. This is typically out of question for phrases of three words or more. However, for $j = 2$ that may not be the case. Bounding the summation with an integral, we get that the expectation is smaller than $1 + V^{1-j/\theta}/(j-1) + O(1/j) = O(n^{\beta(1-\beta j)})$. In that case the total cost of the intersection is $O(n^{\beta(1-\beta j)} \log n)$. Observe that we took j (phrase length) as a constant).

We now point out the times for each type of query, as follows:

- Simple words: A **bin-search** taking $O(\log n)$.
- Phrases of j simple words: the time is $O(j \log n)$, which is both the time to search each word (j **bin-search**es) and to intersect the lists (**lst-inters**). For $j = 2$ the time can be $O(n^{\beta(1-2\beta)} \log n)$, which is $o(n^\beta)$.

– Extended patterns, regular expressions and approximate word matching: **seq-search** and **lst-merge**, taking $O(n^\beta + pn^\alpha \log p)$, where p is the number of matching words in the sequential search phase. This is good if we assume that there is a small number p of matches, which is the case of interest as we explain later.
– Phrases formed with complex patterns: j searches as those for the above expressions (which add up $O(j(n^\beta + pn^\alpha \log p)))$, plus a **lst-inters**, which costs $O(pn^\alpha j \log j)$. Observe that the lists have length $O(pn^\alpha)$. This is because they come from merging the lists of p random words in the vocabulary (which matched the extended or approximate pattern), and therefore the merging-like algorithm is used. The total time is thus $O(j(n^\beta + pn^\alpha \log(pj)))$.
– Approximate phrase matching of j words and k errors: j **seq-search**es considering a number of different **lst-inters**, has a cost of $O\left(j(n^\beta + pn^\alpha \log p) + \binom{k+j}{j} pn^\alpha j \log j\right)$, under the same conditions as above. A number of intersections are made here, since for each word and each number of errors up to k, a different list is kept with the words that matched with each number of errors. Later, we have to try all alternatives of selecting a list of each word, such that the total number of errors does not exceed k. That is $\binom{k+j}{j}$. Again, the lists of occurrences are $O(pn^\alpha)$ size.

In all these results, we can consider j and k as small constants. On the other hand, as shown in [BYN97], $p = O(n^\gamma)$ for γ in the range 0.1 .. 0.2 if the precision is reasonably good (i.e. $k \leq 3$ for words). The same should happen for complex patterns, since otherwise a large p value means a query which has too low precision and is of no use to the final user. This condition can be detected beforehand.

This article was processed using the LaTeX 2_ε macro package with CUP_CS class