Block Addressing Indices for Approximate Text Retrieval [*]

Ricardo Baeza-Yates   Gonzalo Navarro

Department of Computer Science
University of Chile
Blanco Encalada 2120 - Santiago - Chile
{rbaeza,gnavarro}@dcc.uchile.cl

Abstract

Although the issue of approximate text retrieval is gaining importance in the last years, it is currently addressed by only a few indexing schemes. To reduce space requirements, the indices may point to text blocks instead of exact word positions. This is called "block addressing". The most notorious index of this kind is *Glimpse*. However, block addressing has not been well studied yet, especially regarding approximate searching.

Our main contribution is an analytical study of the space-time trade-offs related to the block size. We find that, under reasonable assumptions, it is possible to build an index which is simultaneously sublinear in space overhead and in query time. We validate the analysis with extensive experiments, obtaining typical performance figures. These results are valid not only for approximate searching queries but also for classical ones.

Finally, we propose a new strategy for approximate searching on block addressing indices, which we experimentally find 4-5 times faster than *Glimpse*. This algorithm takes advantage of the index even if the whole text has to be scanned. As a side effect, we find that using blocks of fixed size is better than, say, addressing files.

1 Introduction

With the advent of larger and larger databases, approximate text retrieval is becoming an important issue. As text databases grow and become more heterogeneous, including data from different sources, it is more and more difficult to assure the quality of the information. Errors coming from misspelling, mistyping or from optical character recognition (OCR) are examples of agents that inevitably degrade the quality of large text databases. Words which are stored erroneously are no longer retrievable by means of exact queries. Moreover, even the queries may contain errors, for instance in the case of misspelling a foreign name or missing an accent mark.

A model which captures very well that kind of errors is the *Levenshtein distance*, or simply *edit distance*. The *edit distance* between two strings is defined as the minimum number of character insertions, deletions and replacements needed to make them equal. For example, the edit distance between `"color"` and `"colour"` is 1, while between `"survey"` and `"surgery"` is 2. Phonetic issues can be incorporated in this similarity measure. There are other similarity measures, such as semantic similarity, which are out of the scope of this paper.

The problem of *approximate string matching* is defined as follows: given a text and a pattern, retrieve all the segments (or "occurrences") of the text whose *edit distance* to the pattern is at most k (the number of allowed "errors"). This problem has a number of other applications, such as computational biology, signal processing, etc. We call n the size of the text.

There exist a number of solutions for the on-line version of this problem (i.e. the pattern can be preprocessed but the text cannot). All these algorithms traverse the whole text. If the text database is large, even the fastest on-line algorithms are not practical, and preprocessing the text becomes mandatory. This is normally the case in information retrieval (IR). However, the first indexing schemes for this problem are only a few years old.

There are two types of indexing mechanisms: word-retrieving and sequence-retrieving. In the first one, oriented to natural language text and IR, the index is capable of retrieving every *word* whose edit distance to the pattern is at most k. In the second one, useful also when the text is not natural language, the index can retrieve every matching *sequence*, without notion of word separation.

The existing indices of the first kind are modifications of the inverted list approach. They store the *vocabulary* of the text (i.e. the list of distinct words) and the *occurrences* of each word (i.e. the positions in the text). To search an approximate pattern in the text, the vocabulary is first sequentially scanned, word by word (with an on-line algorithm). Once the set of matching words is known, their positions in the text are retrieved. Since the vocabulary is very small compared to the text, they achieve acceptable performance. These indices can only retrieve whole words or phrases. However, this is in many cases exactly what is wanted. Examples of these indices are *Glimpse* [5] and *Igrep* [1]. *Glimpse* uses *block addressing* (i.e. pointing to blocks of

[*]This work has been supported in part by Fondecyt grants 1960881 and 1950622.

text instead of words) to reduce the size of the index, at the expense of more sequential processing at query time.

This work is focused on block addressing for word retrieving indices. Not only there exist few indexing schemes, but also the problem is not very well studied. However, our main results apply to classical queries too. We study the use of block addressing to obtain indices which are sublinear in space and in query time, and show analytically a range of valid combinations to achieve this. Ours is an average case analysis which gives "big-O" (i.e. growth rate) results and is strongly based on some heuristic rules widely accepted in IR. We validate this analysis with extensive experiments, obtaining typical performance figures.

We also propose a new strategy for approximate searching on block addressing indices, which we experimentally find 4-5 times faster than *Glimpse*, and that unlike *Glimpse*, takes advantage of the vocabulary information even when the whole text has to be verified. As a side effect, we find that block addressing is much better than file addressing (i.e. pointing to files).

This paper is organized as follows. In Section 2 we review previous work. In Section 3 we study analytically the space-time trade-offs related to the block size. In Section 4 we validate experimentally the analysis. In Section 5 we explain our new search algorithm and compare it experimentally against *Glimpse*. Finally, in Section 6 we give our conclusions and future work directions.

2 Previous Work

The first proposal for an approximate word-retrieving index is due to Manber and Wu [5]. In a very practical approach, they propose a scheme based on a modified inverted file and sequential approximate search.

The index structure is as follows: the text is logically divided into "blocks". The index stores all the different words of the text (the vocabulary). For each word, the list of the blocks where the word appears is kept. See Figure 1.

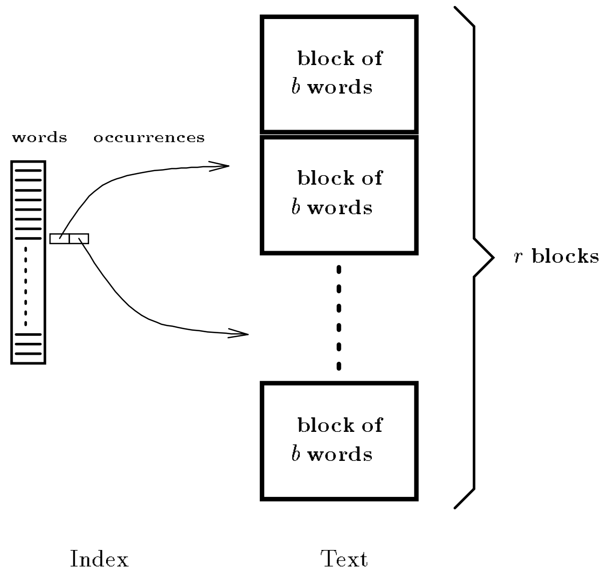


Figure 1: The word indexing scheme.

To search a word allowing errors, the vocabulary is sequentially scanned, word by word, with *Agrep* [9]. *Agrep* is an on-line approximate search software, which will treat the vocabulary as a simple piece of text. For each matching word, all the blocks where it appears in the text are marked. Then, for every marked block (i.e. where some matching word is present), a new sequential search is performed over that block (using *Agrep* again).

The idea of sequentially traversing the vocabulary (which is typically small) leads to a great deal of flexibility in the supported query operations.

The use of blocks makes the index small, at the cost of having to traverse parts of the text sequentially. The index is small not only because the pointers to the blocks may need less bytes, but also because all the occurrences of a word in a single block are referenced only once. This scheme works well if not too many blocks are searched, otherwise it becomes similar to sequential search on the text using *Agrep*. If the number of allowed errors in the pattern is reasonable (1–3), the number of matching words is small. Otherwise the query is of little use in IR terms, because of low precision.

Glimpse does not allow to tune the number or size of the blocks to use. The basic scheme works with 200 to 250 blocks, and works reasonably well for text collections of up to 200 Mb. To cope with larger texts, it offers an index addressing files instead of blocks ("file addressing"). Finally, for very large texts it can be switched to full inversion (i.e. word addressing), where each word points to its exact occurrences in the text. Typical figures for the size of the index with respect to the text are: 2-4% for blocks, 10-15% for files, 25-30% for words. Note that the last percentage is similar to the overheads of classical inverted lists.

To overcome the need of sequentially searching parts of the text, the approach of full inversion is taken in *Igrep* [1]. For each word, the list of all its occurrences in the text are kept.

This changes completely the second phase of the search: once the matching words in the vocabulary are identified, all their occurrence lists are merged and the text is never accessed. This makes the approach much more resistant to the size of the text collection. The scheme is shown to work well with text collections of more than 1 Gb of text.

Instead of *Agrep*, the scheme uses another on-line search algorithm [2], which is especially well suited for short patterns (e.g. words). The scheme also allows to search phrases, by splitting them into words and combining the results.

The index is built in a single pass over the text and in linear time. The construction proceeds in-place, in the sense that the space requirement to build the index is that of the final index.

The analysis shows that the retrieval costs are sublinear for useful searches (i.e. those with reasonable precision). The space requirements are similar to those of classical inverted lists (30-40%). This is the cost for not accessing the text at all, but compression schemes are being studied [6].

3 Average Space-Time Trade-offs

Glimpse and *Igrep* are two extremes of a single idea. *Glimpse* achieves small space overhead at the cost of sequentially traversing parts of the text. *Igrep* achieves better performance by keeping a large index. We study in this section the possibility of having an intermediate index, which is sublinear in size and query performance at the same time. We show that this is possible in general, under reasonable assumptions.

Our analysis is strongly based on some widely accepted heuristic rules which are explained next. Those rules deal with average cases of natural language text. We obtain results regarding the growth rate of index sizes and query times, and therefore our analysis uses the $O()$, $\Omega()$ and $\Theta()$ notation. Hence, our analysis refers to the average case, gives "big-O" results and is valid only if the heuristic rules that we use are valid in the text data.

3.1 Modeling the Text

We assume some empirical rules widely accepted in IR, which are shown accurate in our experiments.

The first one is Heaps law, which relates the text size n and the average vocabulary size V by the formula $V = O(n^\beta)$ for $0 < \beta < 1$ [4].

The second rule is the generalized Zipf's law [10], which states that if the words of the vocabulary are sorted in decreasing order of frequency, then the frequency of the i-th word is $n/(i^\theta H_V^{(\theta)})$, where $H_V^{(\theta)} = \sum_{j=1}^{V} 1/j^\theta$, for some $\theta \geq 1$. For $\theta = 1$ it holds $H_V^{(1)} = \ln V + O(1)$, while for $\theta > 1$ we have $H_V^{(\theta)} = O(1)$. For instance, in the texts of the TREC collection [3], β is between 0.4 and 0.6, while θ is between 1.7 and 2.0.

The third rule assumes that user queries distribute uniformly in the vocabulary, i.e. every word in the vocabulary can be searched with the same probability. This is particularly true in approximate searching, since even if the user searches, say, very unfrequent words, those words match with k errors with other words, with no relation to the frequencies of the matched words.

Finally, the words are assumed to be uniformly distributed in the text. Although widely accepted, this rule may not be true in practice, since words tend to appear repeated in small areas of the text. For our purposes, uniform distribution is a pessimistic assumption we make.

Recall Figure 1. The text of n words is divided into r blocks of size b (hence $n \approx rb$). The vocabulary (i.e. every different word in the text) is stored in the index. For each word, the list of blocks where it appears is stored.

3.2 Query Time Complexity

To search an approximate pattern, a first pass runs an online algorithm over the vocabulary. The sets of blocks where each matching word appears are collected. For each such block, a sequential search is performed on that block.

The sequential pass over the vocabulary is linear in V, hence it is $\Theta(n^\beta)$, which is sublinear in the size of the text.

An important issue is how many words of the vocabulary match a given pattern with k errors. In principle, there is a constant bound to the number of distinct words which match a given pattern with k errors, and therefore we can say that $O(1)$ words in the vocabulary match the pattern. However, not all those words will appear in the vocabulary. Instead, while the vocabulary size increases, the number of matching words that appear increases too, at a lower rate[1]. We show experimentally in the next section that a good model for the number of matching words in the vocabulary is $O(n^\alpha)$ (with $\alpha < \beta$).

For classical word queries we have $\alpha = 0$ (i.e. only one word matches). For prefix searching, regular expressions and other multiple-matching queries, we conjecture that the set of matching words grows also as $O(n^\alpha)$ if the query is going to be useful in terms of precision. However, this issue deserves a separate study and is out of the scope of this paper.

Since the average number of occurrences of each word in the text is $n/V = \Theta(n^{1-\beta})$, the average number of occurrences of the pattern in the text is $O(n^{1-\beta+\alpha})$. This fact is surprising, since one can think in the process of traversing the text, where each word appears with a fixed probability and therefore there is a fixed probability of matching each new word. Under this model the number of matching words is a linear proportion of the text. The fact that this is not the case (demonstrated experimentally in the next section) shows that this model is not realistic. The new words that appear as the text grows deviate it from the model of words appearing with fixed probability.

The blocks to work on in the text are those including some occurrence of the pattern. Call w the number of matching words in the text. Since we assume that the words appear uniformly in the text, the probability that a given block does *not* contain a given word is $1 - 1/r$. Therefore, the probability of not containing any words is $(1-1/r)^w$. Since there are r blocks, the average number of non-empty blocks is $r(1-(1-1/r)^w)$. Since sequentially traversing each block costs $\Theta(b)$, we have a total cost of $\Theta(br(1-(1-1/r)^w)) = \Theta(n(1-(1-1/r)^w))$.

We now simplify that expression. Notice that $(1-1/r)^w = e^{w \ln(1-1/r)} = e^{-w/r + O(w/r^2)} = \Theta(e^{-w/r})$. Hence the query time complexity if w occurrences are searched is

$$\Theta(n(1-e^{-w/r}))$$

As we observed before, $w = \Theta(n^{1-\beta+\alpha})$ on average. Therefore, on average the search cost is $\Theta(n(1-e^{-b/n^{\beta-\alpha}}))$.

Expressions of the form "$1-e^{-x}$" appear frequently in this analysis. We observe that they are $O(x)$ whenever $x = o(1)$ (since $e^{-x} = 1 - x + O(x^2)$). On the other hand, if $x = \Omega(1)$, then e^{-x} is far away from 1, and therefore "$1-e^{-x}$" is $\Omega(1)$.

For the search cost to be sublinear, it is thus necessary that $b = o(n^{\beta-\alpha})$, which we call the "condition for time sublinearity".

3.3 Space Complexity

We consider space now. The average size of the vocabulary itself is already sublinear. However, the total number of references to blocks where each word appears may be linear (it is truly linear in the case of full inversion, which corresponds to single-word blocks, i.e. $b = 1$).

Suppose that a word appears ℓ times in the text. The same argument used above shows that it appears in $\Theta(r(1-$

[1]This is the same phenomenon observed in the size of the vocabulary. In theory, the total number of words is finite and therefore $V = O(1)$. But in practice that limit is never reached, and the model $V = O(n^\beta)$ describes reality much better.

$e^{-\ell/r}$)) blocks on average. Recall that the index stores an entry in the list of occurrences for each different block where a word appears. Under the Zipf's law, the number of occurrences of the i-th most frequent word is $\ell_i = n/(i^\theta H_V^{(\theta)})$. Therefore, the number of blocks where it appears is

$$\Theta\left(r\left(1-e^{-\ell_i/r}\right)\right) = \Theta\left(r\left(1-e^{-b/(i^\theta H_V^{(\theta)})}\right)\right)$$

and the total number of references to blocks is

$$r \sum_{i=1}^{V} 1 - e^{-b/(i^\theta H_V^{(\theta)})} \qquad (1)$$

a summation which is hard to solve exactly. However, we can still obtain the required information of order. We show now that there is a threshold a such that

$$a = \left(\frac{b}{H_V^{(\theta)}}\right)^{1/\theta}$$

1. The $O(a)$ most frequent words appear in $\Theta(r)$ blocks, and therefore contribute $\Theta(ar)$ to the size of the lists of occurrences. This is because each term of the summation (1) is $\Omega(1)$ provided $b = \Omega\left(i^\theta H_V^{(\theta)}\right)$ which is equivalent to $i = O(a)$.
2. The $O(V-a)$ least frequent words appear nearly each one in a different block, that is, if the word appears ℓ times in the text, it appears in $\Omega(\ell)$ blocks. This is because $r(1-e^{-\ell/r}) = \Theta(\ell)$ whenever $\ell = o(r)$. For $\ell_i = n/(i^\theta H_V^{(\theta)})$, this is equivalent to $i = \omega(a)$.

   Summing the contributions of those lists and bounding with an integral we have

$$\sum_{i=a+1}^{V} \frac{n}{i^\theta H_V^{(\theta)}} = \frac{n}{H_V^{(\theta)}} \frac{1/a^{\theta-1} - 1/V^{\theta-1}}{\theta - 1}(1+o(1))$$
$$= \Theta\left(\frac{n}{a^{\theta-1}}\right) = \Theta(ar)$$

   where we realistically assume $\theta > 1$ (we cover the case $\theta = 1$ later).

Therefore, the total space for the lists of occurrences is always $\Theta(ar) = \Theta(rb^{1/\theta})$ for $\theta > 1$.

Hence, for the space to be sublinear we just need $r = o(n)$. Therefore, simultaneous time and space complexity can be achieved whenever $b = o(n^{\beta-\alpha})$ and $r = o(n)$.

3.4 Combined Sublinearity

To be more precise, assume we want to spend

$$\mathit{Space} = \Theta(n^\gamma)$$

space for the index. Given that the vocabulary alone is $O(n^\beta)$, $\gamma \geq \beta$ must hold. Solving $rb^{1/\theta} = n^\gamma$ we have

$$r = \Theta\left(n^{\frac{\gamma\theta-1}{\theta-1}}\right)$$

(a condition which imposes $\gamma \geq 1/\theta$) and

$$b = \Theta\left(n^{\frac{\theta(1-\gamma)}{\theta-1}}\right)$$

Space	Time
$n^{0.90}$	$n^{0.99}$
$n^{0.92}$	$n^{0.95}$
$n^{0.94}$	$n^{0.91}$
$n^{0.96}$	$n^{0.87}$
$n^{0.98}$	$n^{0.82}$
$n^{1.00}$	$n^{0.78}$

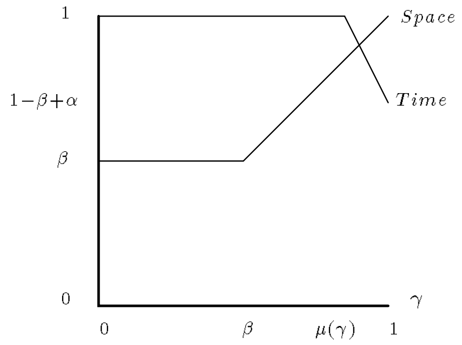


Figure 2: On the left, valid combinations for time and space complexity assuming $\theta = 1.87$, $\beta = 0.4$ and $\alpha = 0.18$. On the right, time and space complexity exponents.

Since the condition for time sublinearity imposes $b = o(n^{\beta-\alpha})$ and we had previously that $\gamma \geq \beta$, we conclude

$$\gamma > \mu(\gamma) = \max\left(\beta, \frac{1+(1-\beta+\alpha)(\theta-1)}{\theta}\right)$$

In that case, the time complexity becomes

$$\mathit{Time} = \Theta\left(n^{1-\beta+\alpha+\frac{\theta(1-\gamma)}{\theta-1}}\right)$$

(and $\Theta(n)$ if $\gamma \leq \mu(\gamma)$). Note that the above expression turns out to be just the number of matching words in the text times the block size.

The combined $\mathit{Time} \times \mathit{Space}$ complexity is

$$\mathit{Time} \times \mathit{Space} = \Theta\left(n^{1-\beta+\alpha+\frac{\theta-\gamma}{\theta-1}}\right)$$

which is minimized for $\gamma = 1$ (full inversion), where $\mathit{Space} = \Theta(n)$ and $\mathit{Time} = \Theta(n^{1-\beta+\alpha})$.

The practical values of the TREC collection show that γ must be larger than 0.77 .. 0.89 in practice, in order to answer queries with at least one error in sublinear time and space. Figure 2 shows possible time and space combinations for $\beta = 0.4$ and $\theta = 1.87$, values that correspond to the collection we use in the experiments. The values correspond to searching with $k = 2$ errors, which, as found in the next section, matches with $\alpha = 0.18$. If less space is used, the time keeps linear (as in *Glimpse*). Better complexities can be expected for larger β (which is normally the case).

We also show schematically the valid time and space combinations. We plot the exponents of n for varying γ. As the plot shows, the only possible combined sublinear complexity is achieved in the range $\mu(\gamma) < \gamma < 1$, which is quite narrow.

We have left aside the case $\theta = 1$, because it is usually not true un practice. However, we show now what happens in this case. The analysis for query time remains the same. For space, we have that $a = \Theta(b/\log V) = \Theta(b/\log n)$. Summing the two parts of the vocabulary we have that the space for the lists of occurrences is

$$\Theta\left(\frac{n}{\log n} + n\left(1 - \frac{\log b}{\log n} + \frac{\log\log n}{\log n}\right)\right)$$

which is sublinear provided $b = \Omega(n^\delta)$, for every $\delta < 1$ (e.g. $b = n/\log n$). This condition opposes to the one for time sublinearity, even for classical searches with $\alpha = 0$. Therefore, it is not possible to achieve combined sublinearity in this (unrealistic) case.

We end this section with a couple of practical considerations regarding this kind of index. First, using blocks of fixed size imposes no penalty on the overall system, since the block mechanism is a logical layer and the files do not need to be physically split or concatenated.

Another consideration that arises is how to build the index incrementally if the block size b has to vary when n grows. Reindexing each time with a new block size is impractical. A possible solution is to keep the current block size until it should be doubled, and then process the lists of occurrences making equal all blocks numbered $2i$ with those numbered $2i+1$ (and deleting the resulting duplicates). This is equivalent to deleting the least significant bit of the block numbers. The process is linear in the size of the index (i.e. sublinear in the text size) and fast in practice.

4 Experimental Validation

In this section we validate experimentally the previous analysis. For our experiments, we use one of the collections contained in TREC, namely the WSJ (Wall Street Journal). The collection contains 250 Mb of text. To mimic common IR scenarios, all the texts were transformed to lower-case, all separators to single spaces (respecting lines); and stop-words were eliminated. We are left with almost 200 Mb of filtered text. Throughout this exposition we talk in terms of the size of the filtered text, which takes 80% of the original text. We measure n and b in bytes, not in words.

The collection is considered as a unique large file, which is logically split into blocks of fixed size. The larger the blocks, the faster to build and the smaller the index, but also the larger the proportion of text to search sequentially at query time. To measure the behavior of the index as n grows, we index the first 20 Mb of the collection, then the first 40 Mb, and so on, up to 200 Mb.

4.1 Vocabulary

We measure V, the number of words in the vocabulary in terms of n (the text size). Figure 3 (left side) shows the growth of the vocabulary. Using least squares we fit the curve $V = 78.81n^{0.40}$. The relative error is very small (0.84%). Therefore, $\beta = 0.4$ for our experiments.

We then measure the number of words that match a given pattern in the vocabulary. For each text size, we select words at random from the vocabulary allowing repetitions. This is to mimic common IR scenarios. In fact, not all user queries are found in the vocabulary in practice, which reduces the number of matches. Hence, this test is pessimistic in that sense.

We test $k = 1$, 2 and 3 errors. To avoid taking into account queries with very low precision (e.g. searching a 3-letter word with 2 errors may match too many words), we impose limits on the length of words selected: only words of length 4 or more are searched with one error, length 6 or more with two errors, and 8 or more with three errors.

We perform a number of queries which is large enough to ensure a relative error smaller than 5% with a 95% confidence interval. Figure 3 (right side) shows the results. We use least squares to fit the curves $0.31n^{0.14}$ for $k = 1$, $0.61n^{0.18}$ for $k = 2$ and $0.88n^{0.19}$ for $k = 3$. In all cases the relative error of the approximation is under 4%. These are the α values mentioned in the analysis.

4.2 Space versus Time for Fixed Block Size

We show the space overhead of the index and the time to answer queries for three different fixed block sizes: 2 Kb, 32 Kb and 512 Kb. See Figure 4. Observe that the time is measured in a machine-independent way, since we show the percentage of the whole text that is sequentially searched. Since the processing time in the vocabulary is negligible, the time complexity is basically proportional to this percentage. The decreasing percentages indicate that the time is sublinear.

The queries are the same used to measure the amount of matching in the vocabulary, again ensuring at most 5% of error with a confidence level of 95%. Using least squares we obtain that the amount of traversed text is $0.10n^{0.79}$ for $b =$ 2 Kb, $0.45n^{0.85}$ for $b = 32$ Kb, and $0.85n^{0.89}$ for $b = 512$ Kb. In all cases, the relative error of the approximation is under 5%. As expected from the analysis, the space overhead becomes linear (since $\gamma = 1$) and the time is sublinear (the analysis predicts $O(n^{0.78})$, which is close to these results).

We observe that the analysis is closer to the curve for smaller b. This is because the fact that $b = O(1)$ shows up earlier (i.e. for smaller n) when b is smaller. The curves with larger b will converge to the same exponents for larger n.

4.3 Space versus Time for Fixed Number of Blocks

To show the other extreme, we take the case of fixed r. The analysis predicts that the time should be linear and the space should be sublinear (more specifically, $O(n^{1/\theta}) = O(n^{0.53})$). This is the model used in *Glimpse* for the tiny index (where $r \approx 256$).

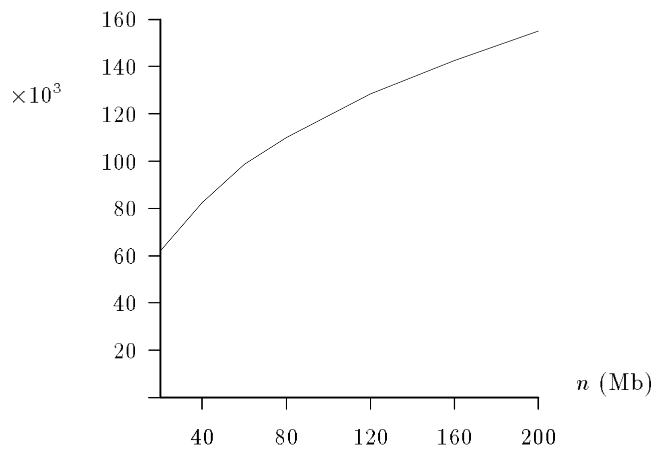
See Figure 5, where we measure again space overhead and query times, for $r = 2^8$, 2^{12} and 2^{16}. Using least squares we find that the space overhead is sublinear in the text size n. For $r = 2^8$ we have that the space is $0.87n^{0.53}$, for $r = 2^{12}$ it is $0.78n^{0.75}$, and for $r = 2^{16}$ it is $0.74n^{0.87}$. The relative error of the approximation is under 3%. As before, the analysis is closer to the curve for smaller r, by similar reasons (the effect is noticed sooner for smaller r).

On the other hand, the percentage of the traversed text increases. This is because the number of matching words increases as $O(n^{1-\beta+\alpha})$. The percentage will eventually stabilize, since it is increasing and bounded by 100%.

4.4 Sublinear Space and Time

Finally, we show experimentally in Figure 6 that time and space can be simultaneously sublinear. We test $\gamma = 0.92$, 0.94 and 0.96. The analysis predicts the values shown in the table of Figure 2.

Using least squares we find that the space overhead is sublinear and very close to the predictions: $0.42n^{0.95}$, $0.41n^{0.92}$



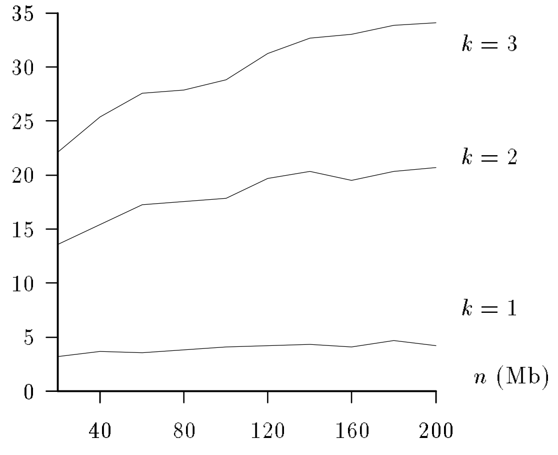
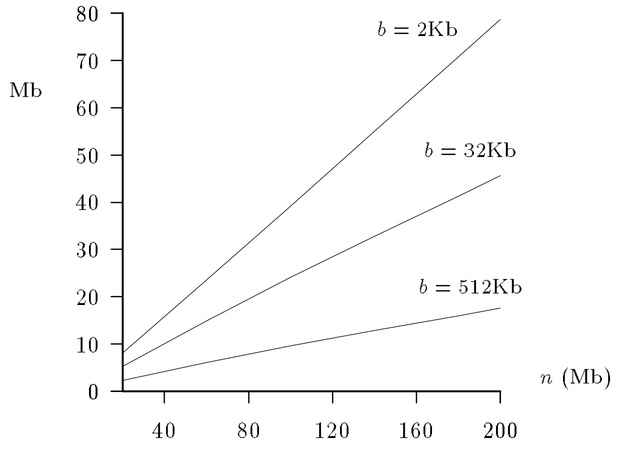


Figure 3: Vocabulary tests for the WSJ collection. On the left, the number of words in the vocabulary. On the right, number of matching words in the vocabulary.



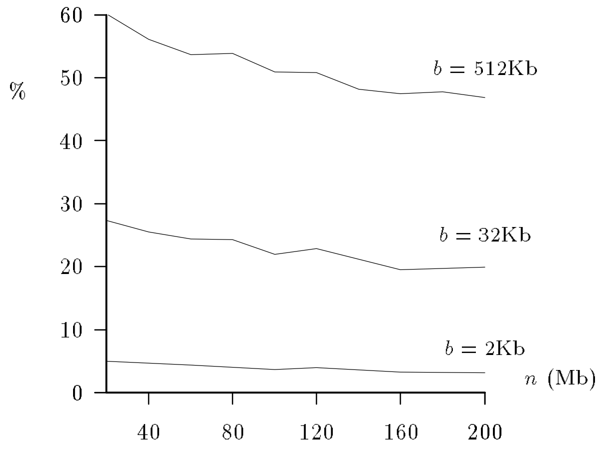


Figure 4: Experiments for fixed block size b. On the left, space taken by the indices. On the right, percentages of the text sizes sequentially searched.

and $0.40n^{0.89}$ respectively. The error of the approximations is under 1%.

The percentage of the traversed text decreases, showing that the time is also sublinear. The least squares approximation shows that the query times for the above γ values are $0.24n^{0.95}$, $0.17n^{0.94}$ and $0.11n^{0.91}$, respectively. The relative error is smaller than 2%.

Hence, we can have for this text[2] an $O(n^{0.94})$ space and time index (our analysis predicts $O(n^{0.93})$).

5 A New Searching Algorithm

We present in this section an alternative approximate search approach which is faster than *Glimpse*.

We also start the search by sequentially scanning the vocabulary with an on-line approximate search algorithm. Once the blocks to search have been obtained, *Glimpse* uses *Agrep* (i.e. approximate searching) again over the blocks. However, this can be done better.

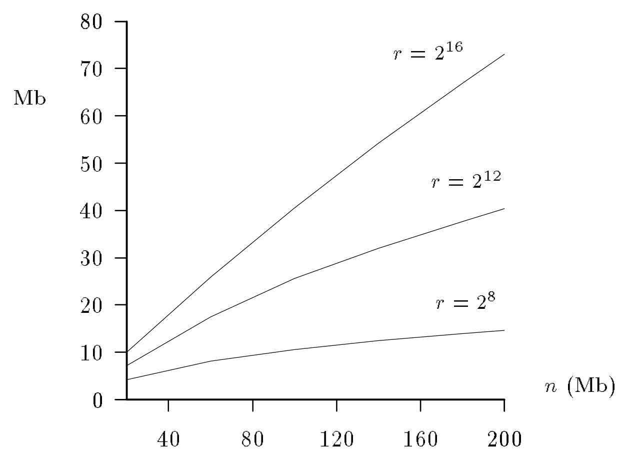
Since we have run an online search over the vocabulary first, we know not only which blocks contain an approximate match of the search pattern, but also which words of the vocabulary matched the pattern and are present in each block. Hence, instead of using again approximate search over the blocks as *Glimpse*, we can run an *exact* search for those matching words found in the vocabulary. In most cases, this can be done much more efficiently than approximate searching. Moreover, since as we show later, most of the search time is spent in the search of the text blocks, this improvement has a strong impact on the overall search time.

We use an extension of the Boyer-Moore-Horspool-Sunday algorithm [7] to multipattern search. This gave us better results than an Aho-Corasick machine, since as shown in Figure 3, few words are searched on each block (this decision is also supported by [8]).

We compared this strategy against *Glimpse* version 4.0. We used the "small" index provided by *Glimpse*, i.e. the one addressing files (i.e. the sequential search must be done on the matching files). Our index used also files as the addressing unit for this comparison.

All the tests were run on a Sun SparcServer 1000 with 128 Mb RAM, running SunOS 5.5. However, only 4 Mb or RAM were used by the indexers. The tests were run when the

[2]The WSJ collection has especially low β value, which worsens the complexities. We selected it for its size.



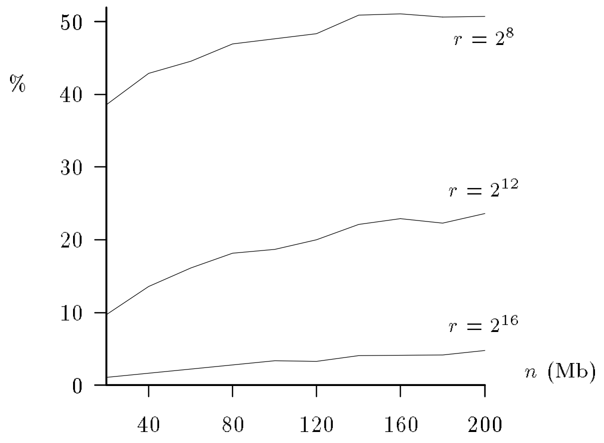
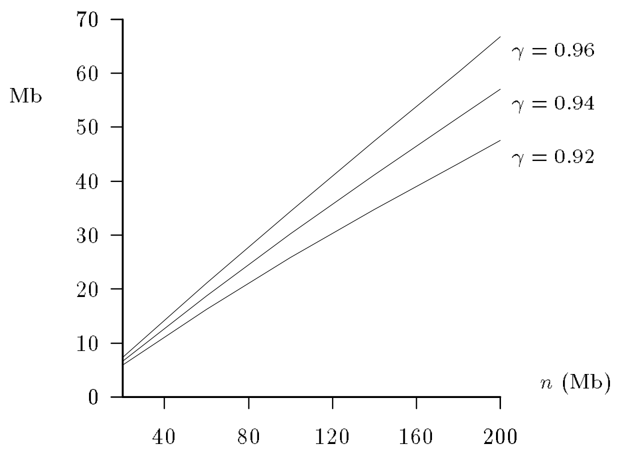


Figure 5: Experiments for fixed number of blocks r. On the left, space taken by the indices. On the right, percentages of the text sizes sequentially searched.



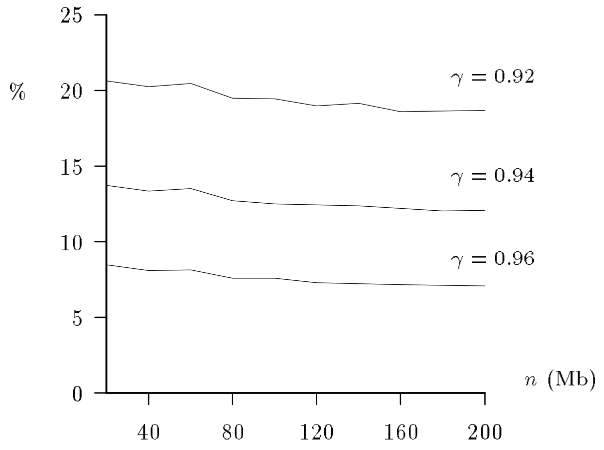


Figure 6: Experiments for fixed γ (simultaneous sublinearity). On the left, space taken by the indices. On the right, percentages of the text sizes sequentially searched.

server was not performing other tasks. We used the same WSJ collection for the comparison. It is composed of 278 files of almost 1 Mb each.

The stop-word mechanism and treatment of upper and lower-case letters is somewhat particular in *Glimpse*. We circumvent this problem by performing all filtering and stop-word elimination directly in the source files, and then using both indices without filtering or stop-words considerations.

Our indexer took near 16 minutes to index the collection (i.e. more than 10 Mb per second), while *Glimpse* took 28 minutes. This is due to different internal details which are not of interest to this work, e.g. the indexers have different capabilities apart from approximate searching. Both indices took approximately 7 Mb. This is less than 3% of the size of the collection (this good result is because the files are quite large).

We are not comparing the complete indexing mechanisms, but only their strategy to cope with approximate search of words when they have to be sequentially searched on the text. Issues such as a different addressing granularity will not change the proportion between the search times.

In both indices we retrieve whole words that match the pattern. This is the default in this paper and we believe that this option is more natural to the final user than allowing subword matching (i.e. `"sense"` matching with *one* error in `"consensus"`).

Table 1 (upper table) shows the times obtained (user times). As it can be seen, the mechanism we propose is 4-5 times faster in practice (i.e. taking into account all the processing needed). We also show the percentage of the text sequentially inspected and the average number of matches found, as well as the number of words matching in the vocabulary. We can see that an important part of the text is inspected, even for queries with acceptable precision (this is because the files are large). Moreover, the times are almost proportional to the amount of sequential search done (we process near 5 Mb/sec, while *Glimpse* is close to 1 Mb/sec). Therefore, the advantage of searching with a multipattern exact search instead of an approximate search algorithm is evident. Even if the whole text is searched (in which case *Glimpse* is not better than *Agrep*, i.e. a complete sequential search), our indexing scheme takes advantage of the vocabulary, because it never searches the text for an approximate pattern.

Table 1 (lower part) presents the less interesting case in which subword matching is also allowed. The precision is much lower (i.e. there are more matches), which shows that this query is unlikely to be interesting for the users. It can also be seen that much more text is traversed. The perfor-

Errors	Ours	*Glimpse*	Ours/*Glimpse*	% inspected	# matches	# vocab. matches
Matching Complete Words						
1	8.20	34.99	23.42%	24.94%	871.86	4.97
2	18.05	82.50	21.91%	43.83%	2591.02	25.54
3	29.37	143.69	20.43%	77.81%	7341.84	31.15
Subword Matching Allowed						
1	39.37	16.05	245.30%	41.45%	44541.50	159.07
2	73.04	64.12	113.91%	64.28%	44991.80	230.48
3	75.84	132.56	57.21%	77.39%	31150.50	182.92

Table 1: Times (in seconds) and other statistics to retrieve all occurrences of a random word with different number of errors.

mance of our algorithm degrades due to a larger amount of words matching in the vocabulary, which reduces the effectiveness of our multipattern exact searching against plain approximate search. On the other hand, *Glimpse* improves for one and two errors because of specialized internal algorithms to deal with this case. The net result is that our algorithm is slower for one and two errors, although it is still faster for three errors. This test shows that our approach is better when not too many words match in the vocabulary, which is normally the case of useful queries.

6 Conclusions and Future Work

We focused on the problem of block addressing for approximate word retrieving indices. We found theoretically and experimentally that it is possible to obtain a block addressing index which is at the same time sublinear in space (like *Glimpse*) and in query time performance (like full inverted indices), and showed practical compromises achieving that goal. For instance, we built for our example text an index which is $O(n^{0.94})$ space and answers approximate queries in $O(n^{0.94})$ time. Those results apply to classical queries too (not only to approximate searching). Finally, we proposed a variation in *Glimpse*'s sequential search, which is 4-5 times faster than *Glimpse* and takes advantage of the index even if all the text must be processed.

We also discovered, as a side effect, that block addressing is much better in general than file addressing, because blocks are of fixed size. This is not shown in the experiments because the files were cut to almost the same size, but on normal files the differences in length make file addressing searching more than twice the text that is searched with block addressing. This translates directly into performance improvements. The reason is that some files are quite long while others are short. Larger files have higher probability of containing a given word and therefore are traversed more frequently. Therefore, the amount to search is directly related to the variance in the size of the addressing units. As explained, using blocks imposes no penalty on the overall system.

We plan to study other improvements to the search algorithm. For instance, we observe that when the vocabulary is sequentially searched, the words are in lexicographical order, and therefore tend to share a common prefix with the previous one. This information could be stored in the index at little space penalty. At query time, common prefixes can be skipped by restarting the processing of the previous word at the proper point.

Finally, we have still not considered efficient storage of the pointers to blocks. In our current implementation, each pointer takes one computer word (in the experiments against *Glimpse*, the small number of files allowed to use two bytes). Clearly, they can be smaller if r is not large (e.g. the tiny index of *Glimpse* uses pointers of one byte because $r < 256$). In theory, this adds a multiplying factor of $O\left(\frac{\log r}{\log n}\right)$ to the index space, which does not affect our analysis of sublinearity. However, the reduction in space may be significative in practice (up to 50% reduction). We are also studying compression schemes to search directly in the compressed text.

References

[1] M. Araújo, G. Navarro, and N. Ziviani. Large text searching allowing errors. In *Proc. 4th South American Workshop on String Processing, WSP'97*, 1997. Valparaíso, Chile. To appear.

[2] R. Baeza-Yates and G. Navarro. A faster algorithm for approximate string matching. In *Proc. CPM'96*, pages 1–23, 1996.

[3] D. Harman. Overview of the Third Text REtrieval Conference. In *Proc. Third Text REtrieval Conference (TREC-3)*, pages 1–19, 1995. NIST Special Publication 500-207.

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

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

[6] E. Moura, G. Navarro, and N. Ziviani. Indexing compressed text. In *Proc. 4th South American Workshop on String Processing, WSP'97*, 1997. Valparaíso, Chile. To appear.

[7] D. Sunday. A very fast substring search algorithm. *CACM*, 33(8):132–142, Aug 1990.

[8] B. Watson. The performance of single and multiple keyword pattern matching algorithms. In *Proc. WSP'96*, pages 280–294, 1996.

[9] S. Wu and U. Manber. Agrep – a fast approximate pattern-matching tool. In *Proc. USENIX*, pages 153–162, 1992.

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