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

The issue of reducing the space overhead when indexing large text databases is becoming more and more important, as the text collections grow in size. Another subject, which is gaining importance as text databases grow and get more heterogeneous and error prone, is that of flexible string matching. One of the best tools to make the search more flexible is to allow a limited number of differences between the words found and those sought. This is called "approximate text searching", which is becoming more and more popular.

In recent years some indexing schemes with very low space overhead have appeared, some of them dealing with approximate searching. These low overhead indices (whose most notorious exponent is *Glimpse*) are modified inverted files, where space is saved by making the lists of occurrences point to text blocks instead of exact word positions. Despite their existence, little is known about the expected behavior of these "block addressing" indices, and even less is known when it comes to cope with approximate search.

Our main contribution is an analytical study of the space-time trade-offs for indexed text searching. We study the space overhead and retrieval times as functions of 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. This surprising analytical conclusion is validated with extensive experiments, obtaining typical performance figures. These results are valid for classical exact queries as well as for approximate searching.

We apply our analysis to the Web, using recent statistics on the distribution of the document sizes. We show that pointing to documents instead of to fixed size blocks reduces space requirements but increases search times.

1 Introduction

One of the most outstanding characteristics of modern textual databases is their impressive sizes. Special-purpose collections such as TREC [18] contains hundreds of gigabytes in its last version. The Web, a gigantic ad-hoc distributed text collection, had more than 1 terabyte estimated in 1998 [5].

Handling text collections of these sizes and being able to efficiently search on them is becoming a complex task [5, 36]. First of all, it is impossible to sequentially search the whole text for the user specified strings of interest. Even the fastest sequential algorithms would need from minutes to hours for answering the simplest queries, not to mention the extra problems of a distributed text connected by relatively slow links like large portions of the Web. Specialized data structures built

*This work has been supported in part by Fondef (Chile) grant 96-1064.

on the text, called *indices*, are used to speed up the search. This causes a space problem, since not only the text has to be stored but also its index, which typically needs from 20% to 200% extra space over the text size.

However, this is not the only problem. Many text databases, like the Web, are heterogeneous and error-prone, since they store data from different sources and there is little or no quality control on them. 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. The last years witnessed different improvements on query flexibility, some of them regarding the type of patterns that can be searched for. From tools as simple as case sensitiveness to the ability to search regular expressions and to allow some errors in the matches, virtually no current commercial text index is limited to simply search for exact patterns. Of course, searching such "extended" patterns is harder than the classical search for simple ones, which demands new indexing techniques to speed up this task.

Some indices dealing with the space problem and with more flexible searching have appeared in recent years, *Glimpse* [25] probably being the best known exponent. Most of them are modified inverted files, based on the concept of *block addressing*: the text is logically split in blocks and the index is able to tell which blocks a word appears in, but not its exact positions. Despite their existence as software systems, little is known about the expected behavior of block addressing indices. Even less is known about their performance regarding searching for extended patterns.

In this work, we study the use of block addressing to obtain indices which are sublinear in space and in query time simultaneously, and show analytically a range of valid block sizes to achieve this. The combined sublinearity means that, as the text grows, the space overhead of the index and the time to answer a query become less and less significant as a proportion of the text size. This surprising result applies to classical queries as well as to queries that allow errors in the matches, which is one of the most important classes of extended patterns. 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 Information Retrieval (IR). We validate these analytical results with extensive experiments, obtaining typical performance figures.

Finally, we apply our analysis to an interesting particular case of document addressing (where the documents are of variable size): we use recently obtained statistics from the distribution of the page sizes in the Web [13] and apply our machinery to determine the space overhead and retrieval time of an index for a collection of Web pages. We show that having documents of different sizes reduces space requirements in the index but increases search times if the documents have to be traversed.

As a side result, we prove new relations between some laws of Information Retrieval which were previously unnoticed. We also study experimentally how many different words from a text match a query when errors are allowed in the matches, and conjecture that a similar rule is followed by other extended patterns.

This paper is organized as follows. In Section 2 we explain the issue of text searching allowing errors. In Section 3 we present inverted files and the concept of block addressing. In Section 4 we study analytically the space-time trade-offs related to the block size. In Section 5 we validate experimentally the analysis. In Section 6 we apply our analysis to the Web statistics. Finally, in Section 7 we give our conclusions and future work directions. A previous partial version of this work appeared in [3].

2 Text Searching Allowing Errors

One of the main goals when searching for extended patterns is to find words whose exact spelling is not known. This encompasses also those text words which are incorrectly written. The problem of correcting misspelled words in written text is rather old. We could find references from the twenties [26], and perhaps there are older ones. By the sixties, a number of ad-hoc models to match incorrectly written versions of a word existed, like those of Blair [10], Damerau [14] and the popular Soundex method, described for instance in [21, 17]. However, some time elapsed until it was realized [34] that such ad-hoc models were inferior[1] to simple variants of the so-called *Levenshtein* (or edit) distance [23, 24].

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 distance, by changing the cost of the different operations. The goal is, then, to find the text words which are close (in the sense of edit distance) to a given pattern.

More formally, the problem of *approximate string matching* (or *string matching allowing errors*) is defined as follows: given a text (of size n) 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 in computational biology, signal processing, etc.

There exist a number of solutions for the on-line version of this problem [31] (i.e. the pattern can be preprocessed but the text cannot). All these algorithms traverse the whole text sequentially. 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 IR. However, the first indexing schemes for this problem are only a few years old.

There are two types of indexing mechanisms that address this problem: sequence-retrieving and word-retrieving. The first ones can retrieve every matching *substring* of the text, and they do not rely on the concept of word. This makes them suitable for applications such as genetic databases. However, the existing indices are rather immature. Almost all are prototypes in an experimental stage and unable to handle large volumes of text. The indices typically take four to twelve times the size of the text. Some examples are [12, 16, 22, 30, 32, 33].

Word-retrieving indices, although less general, are better suited to natural language applications and IR. They rely on the concept of word, and are able of retrieving every *word* whose edit distance to the pattern is at most k. This simplification allows the existence of very efficient implementations. As all them are inverted files with a modified search technique, we defer their discussion to the next section.

[1] Semantic similarity is a different issue, not covered in this paper.

3 Inverted Files and Block Addressing

An inverted index (or file) has two parts: vocabulary and occurrences [15, 5]. The *vocabulary* of the text is the list of distinct words that appear in it. The *occurrences* contain, for each vocabulary word, the list of the text positions where the word appears. A classical query is solved by searching the pattern in the vocabulary (using binary search or an auxiliary data structure) and retrieving the list of its occurrences (i.e. the positions where the pattern appears in the text). Search times are very good, but the best implementations of inverted indices pose a 20% to 30% space overhead over the text size.

3.1 Block Addressing

Block addressing is a technique to reduce the space requirements of an inverted file. It was first proposed in a system called *Glimpse* (see Section 3.3). The idea is that the text is logically divided in blocks, and the occurrences do not point to exact word positions but only to the blocks where the word appears. Space is saved because there are less blocks than text positions (and hence the pointers are shorter), and also because all the occurrences of a given word in a single text block are referenced only once. Figure 1 illustrates a block addressing index with r blocks of size b (i.e. $n = rb$).

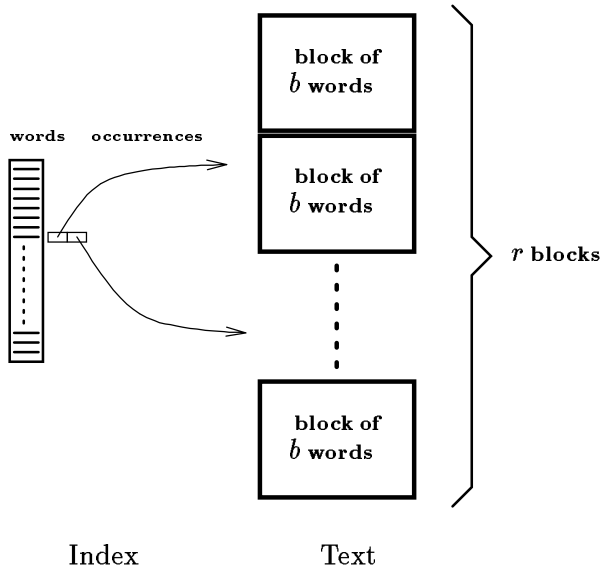

Figure 1: The word indexing scheme.

Searching in a block addressing index is similar to searching in a traditional one (which we call "full inverted index" in this paper). The pattern is searched in the vocabulary and a list of blocks where the pattern appears is retrieved. However, to obtain the exact pattern positions in the text, a sequential search over the qualifying blocks becomes necessary. The index is therefore used as a filter to avoid a sequential search over some blocks, while the others need to be checked. Hence, the reduction in space requirements is obtained at the expense of higher search costs.

At this point the reader may wonder which is the advantage of pointing to artificial blocks instead of pointing to documents (or files), this way following the natural divisions of the text collection. If we consider the case of simple queries (say, one word), where we are required to return only the list of matching documents, then pointing to documents is a very adequate choice. Moreover, as we see later, it may reduce space requirements with respect to using blocks of fixed size. Also, if we use blocks of fixed size and pack many short documents in a logical block, we will have to traverse the matching blocks (even for these simple queries) to determine which documents inside the block actually matched.

However, consider the case where we are required to deliver the exact positions which match a pattern. In this case we need to sequentially traverse the qualifying blocks or documents to find the exact positions. Moreover, in some types of queries such as phrases or proximity queries, the index can only tell that two words appear in the same block, and we need to traverse it in order to determine if they form a phrase.

In this case, pointing to documents of different sizes is not a good idea because larger documents are searched with higher probability and searching them costs more. In fact, the expected cost of the search is directly related to the variance in the size of the pointed documents. This suggests that if the documents have different sizes it may be a good idea to (logically) partition large documents into blocks and to put together small documents, such that blocks of the same size are used.

3.2 Searching Allowing Errors

An inverted file can be easily converted into an efficient word-retrieving index for approximate string matching. This idea, again, was firstly proposed for *Glimpse* (see Section 3.3). To search an approximate pattern in the text, we start by sequentially scanning the vocabulary, word by word, using an on-line algorithm. This allows to collect the set of different words that match the query. Once these words are known, their positions in the text are retrieved and all the lists are merged into a single one, which is the final answer.

This scheme works well because the vocabulary grows slowly as the text grows, and in large text collections it takes less than 1% of the text size. This well-known phenomenon in IR, called Heaps' law [19], is described in detail later in this paper.

When combined with block addressing, the result is a two-stage sequential search process. First, the vocabulary is sequentially searched and the list of qualifying blocks is obtained. Second, each such block is sequentially traversed to find the actual matches. The second step, as explained, may be absent if we point to documents, search for a single word, and want only the list of qualifying documents.

Notice that this idea can be used not only for approximate searching, but also to search for any extended pattern, as long as words in the pattern are matched to words in the text.

3.3 Representative Systems

Although pointing to documents is an old practice, using blocks of fixed size was first proposed in a searching software called *Glimpse*, due to Manber and Wu [25]. In a very practical approach, they present a scheme combining a block addressing inverted file and the ability to search for extended patterns (which includes approximate searching).

To search an extended pattern, the vocabulary is sequentially scanned, word by word, with *Agrep* [37]. *Agrep* is an efficient on-line text searching software, which treats 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 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. This scheme works well if not too many blocks are searched, otherwise it becomes similar to sequentially searching 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 its low precision.

Glimpse has a limited flexibility regarding the number or size of the blocks to use. The basic scheme (called the *tiny* tindex) uses 200 to 250 blocks, and works reasonably well for text collections of up to 200 Mb. As the text grows more, the blocks become too large and almost always match the query, which converts the search into a sequential scan over the whole text. To cope with larger texts, *Glimpse* offers an index addressing files instead of blocks ("file addressing"), which is called the *small* index. 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. This is called the *medium* index. Typical figures for the index size with respect to the text are: 2-4% for blocks (tiny), 10-15% for files (small), 25-30% for words (medium). The last percentage is similar to the overheads of classical full inverted files.

As an example of full inversion we mention a recent one, *Igrep* [2], which inherits from *Glimpse* the ability of (and the technique used for) searching extended patterns and allowing errors. Since every text word is referenced in the list of occurrences, the index poses a fixed overhead over the text size, close to 30-35% in this case. Since the sequential traversal over the blocks is not necessary, the text is never accessed and the approach is much more resistant to the size of the text collection.

Apart from those of *Glimpse*, the index is enriched with further capabilities to search phrases of extended patterns and to search a complete phrase allowing a given number of errors across the whole phrase. The index is built in a single pass over the text using an in-place construction.

A detailed analysis of the search times for different types of simple and extended patterns is presented in [2]. The analysis shows that the retrieval costs are sublinear for useful searches (i.e. those with reasonable precision).

We cannot finish this section without mentioning compression. Compressing the text and/or the inverted file is an orthogonal technique to reduce the overall space usage. This trend is also rather new, since compression and searching have been traditionally regarded as exclusive tasks [7].

New text compression techniques have been studied to make the compressed text suitable of being accessed randomly, and decompressed quickly if needed [27]. These abilities are essential to be used in a text retrieval system. The scheme uses a Huffman coding [20] on words, where the words instead of the characters are the symbols to be coded. Since the words are the symbols of the coder, a table with all the words (that is, a vocabulary) is stored together with the compressed file. On top of this technique, two lines of development have emerged.

A first one was implemented in a system called *MG* [9, 36, 8], which is a compressed inverted file that indexes compressed text. The index points to document and does not search the text directly. Only the qualifying documents are decompressed. The authors show that the overall efficiency is maintined despite the decompression effort, and sometimes even improved thanks to reduced I/O efforts. Note that the inverted file can be integrated with the compressed text since both use the text vocabulary.

A second line of development is direct searching on compressed text (without decompressing). This has been proposed in a software called *Cgrep* [29, 28]. *Cgrep* is a compressor able to efficiently search from simple patterns to regular expressions, allowing or not errors. It is based on Huffman coding on words. The search starts in the vocabulary, much as in inverted indices. Once the matching words are obtained, their compressed codes are searched in the compressed text. Although there is no indexing and all the text has to be traversed, the search is a multipattern exact search for the compressed codes of the matching words, which can be much faster than the original search (e.g. an approximate search). This allows *Cgrep* to be faster than *Agrep*.

Finally, there exists undergoing work to integrate *Cgrep* into a block addressing compressed index, much like *MG* except for the extended patterns handled and because some text blocks have to be searched (with *Cgrep*). Index and text compression reduce the effective size of the text collections to handle, but unlike block addressing, they cannot make it sublinear if we assume constant entropy in the text.

4 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 $O()$, $o()$, $\Omega()$, $\omega()$ and $\Theta()$ notation. We recall that they correspond, respectively, to the signs $\leq$, $<$, $\geq$, $>$ and $=$ with respect to the growth rate of the functions[2].

[2]More formally, $f(n)$ is $O(g(n))$ iff there exist $n_0, c > 0$ such that for all $n > n_0$, $f(n) \leq c \times g(n)$. $f(n)$ is $\Omega(g(n))$ iff $g(n)$ is $O(f(n))$. $f(n)$ is $\Theta(g(n))$ if $f(n)$ is $O(g(n))$ and $\Omega(g(n))$. $f(n)$ is $o(g(n))$ if $f(n)$ is not $\Omega(g(n))$. $f(n)$ is $\omega(g(n))$ if $f(n)$ is not $O(g(n))$.

4.1 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 [19], which relates the text size n and the average vocabulary size V by the formula $V = \Theta(n^\beta)$ for $0 < \beta < 1$.

The second rule is the generalized Zipf's law [38], 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 $H_V^{(\theta)} = O(1)$.

Something which is not said in the literature and that is our first contribution is that these two rules can be related. Assume that the least frequent word appears $O(1)$ times in the text (this is more than reasonable in practice, since a large number of words appear only once). Since there are $\Theta(n^\beta)$ different words, then the least frequent word has rank $i = \Theta(n^\beta)$. The number of occurrences of this word is, by Zipf's law,

$$\frac{n}{i^\theta H_V^{(\theta)}} \quad = \quad \Theta\left(\frac{n}{n^{\beta\theta} H_V^{(\theta)}}\right)$$

and this must be $O(1)$. This implies that, as n grows, $\beta = 1/\theta$. This equality may not hold exactly for real collections. This is because the relation is asymptotical and hence is valid for sufficiently large n, and because Heaps' and Zipf's rules are approximations. For instance, in the texts of the TREC-2 collection [18] (described in Section 5), β is between 0.4 and 0.6, while θ is between 1.7 and 2.0. Considering each collection separately, $\beta\theta$ is between 0.80 and 1.04.

We point out now other assumptions we make. We assume that user queries distribute uniformly in the vocabulary, i.e. every word in the vocabulary can be searched with the same probability. This is not true in practice, since unfrequent words are searched with higher probability (see some tentative models in [11]). On the other hand, approximate searching makes this distribution more uniform, since unfrequent words may match match with k errors with other words, with little relation to the frequencies of the matched words. In general, however, the assumption of uniform distribution in the vocabulary is pessimistic.

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. Uniform distribution in the text is another pessimistic assumption we make, since more blocks are traversed when the distribution is uniform.

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.

4.2 Query Time Complexity

To search an approximate pattern, a first pass runs an on-line 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[3]. 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.

This observation is our second contribution. Notice that the root of this fact lies in the restriction imposed by these indices, stating that words must be matched completely instead of allowing any text substring. If we allowed to match the pattern against any text substring, then the matching probability would be clearly uniform, and sublinear retrieval time should be impossible if we had to collect all the matches (since there would be $\Theta(n)$ matches). However, as we force to match against complete words, the fact that more and more different words are appearing makes the matching probability to be reduced as the text grows, and the total number of matches is sublinear in the text size. This is a basic limitation of sequence-retrieving indices with respect to word-retrieving indices.

The blocks to work on in the text are those including some (exact or approximate) occurrence of the pattern. We model the process as follows: an approximate search first selects $O(n^\alpha)$ random words from the vocabulary, which is of size $\Theta(n^\beta)$. Hence, the probability of a given vocabulary word to be selected by the search is $O(n^{\alpha-\beta})$. To determine whether a block needs to be searched or not, imagine that we take each one of its b words and look whether it is selected in the vocabulary. We work on the block if any of its words has been selected in the vocabulary.

The probability of a word from the block to be selected is $O(n^{\alpha-\beta})$. The probability that none of the words in the block is selected is therefore $(1 - O(n^{\alpha-\beta}))^b$. The total amount of work is

[3] This is the same phenomenon observed in the size of the vocabulary. In theory, the total number of English 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.

obtained by multiplying the number of blocks (r) times the work to do per selected block (b) times the probability that some word in the block is selected. This is

$$\Theta\left(rb\left(1-\left(1-n^{\alpha-\beta}\right)^{b}\right)\right) \quad = \quad \Theta\left(n\left(1-e^{-\Theta(b/n^{\beta-\alpha})}\right)\right) \tag{1}$$

where for the last step we used that $(1-x)^y = e^{y\ln(1-x)} = e^{y(-x+O(x^2))} = e^{-\Theta(yx)})$ provided $x = o(1)$.

We are interested in determining in which cases the above formula is sublinear in n or not. Expressions of the form "$1-e^{-x}$" appear a couple of times 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". When this condition holds, we derive from Eq. (1) that

$$Time \quad = \quad \Theta\left(n^{\beta} \; + \; bn^{1-\beta+\alpha}\right) \tag{2}$$

where we recall that the first n^{β} in the time formula corresponds to the vocabulary traversal.

4.3 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. $b = 1$).

The analysis is very simple if we notice that each block of size b has $O(b^{\beta})$ different words, by Heaps' law. Each different word that appears in each different block will correspond to a different entry in the inverted index. Hence, the size of this index is just the number of different words of each block times the number of different blocks, that is, $O(rb^{\beta})$. Hence, for the space to be sublinear we just need $r = o(n)$, or equivalently, $b = \omega(1)$.

However, we have assumed the validity of an asymptotic rule such as Heaps' law for blocks, which are much smaller than the whole text. As we show in the experiments, the rule is still valid but the β of the blocks converges to its definitive value when the blocks are larger than 1 Mb. In the Appendix we redo this analysis using the Zipf's law and reasoning with the whole collection, which is much more complex. The result is $O(rb^{1/\theta})$. This confirms that for both rules to be valid it must hold $\beta = 1/\theta$. In the analysis that follows, β and $1/\theta$ can be used interchangeably whenever the space complexity is involved. In particular, we use $1/\theta$ to draw the actual numbers, since it is more precise. So we have

$$Space \quad = \quad \Theta\left(rb^{\beta}\right) \quad = \quad \Theta\left(rb^{1/\theta}\right) \tag{3}$$

4.4 Combined Sublinearity

Simultaneous time and space complexity can be achieved whenever $b = o(n^{\beta-\alpha})$ and $r = o(n)$. To be more precise, assume we want to spend

$$Space \quad = \quad \Theta\left(n^{\gamma}\right)$$

space for the index. Given that the vocabulary alone is $O(n^\beta)$, $\gamma \geq \beta$ must hold. Solving $rb^\beta = n^\gamma$ with Eq. (3) we have

$$r \;=\; \Theta\left(n^{\frac{\gamma-\beta}{1-\beta}}\right) \quad , \quad b \;=\; \Theta\left(n^{\frac{1-\gamma}{1-\beta}}\right)$$

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

$$\gamma \;>\; \mu(\gamma) \;=\; 1-(1-\beta)(\beta-\alpha)$$

(which implies $\gamma \geq \beta$). In that case, the time complexity (computed using Eq. (2)) becomes

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

(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.

Note that $\gamma = 1$ corresponds to full inversion, where $Space = \Theta(n)$ and $Time = \Theta(n^{1-\beta+\alpha})$. This is the search complexity of the *Igrep* software [2]. On the other extreme, the "tiny" index of *Glimpse* corresponds to $\gamma = \beta$, in which case $Time = \Theta(n)$ and $Space = \Theta(n^\beta)$, i.e. just the necessary to store the vocabulary. None of these two extremes achieve simultaneous sublinearity.

The practical values of the TREC collection (described in Section 5) 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, has $\alpha = 0.18$. If less space is used, the time keeps linear (as in *Glimpse*). If we consider classical exact queries, then $\alpha = 0$ (since only one word matches in the vocabulary), and then it is possible, for instance, to have an index which is $O(n^{0.85})$ in space and retrieval time.

The figure also shows 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.

It is interesting also to consider for which γ we obtain an index which is $O(n^\gamma)$ space and time (i.e. both complexities are equal). This gives a good measure of what can be achieved in combined sublinearity. The resulting formula is

$$\frac{1+(1-\beta+\alpha)(1-\beta)}{2-\beta}$$

which, assuming β between 0.4 and 0.6, ranges from 0.83 to 0.85 for classical queries ($\alpha = 0$) and from 0.89 to 0.93 for reasonable approximate queries ($\alpha = 0.2$). It is interesting that this measure is not monotonous with respect to β, achieving the optimum for $\beta = 2 - \sqrt{2-\alpha}$ (e.g. $\beta = 0.59$ for $\alpha = 0$ or $\beta = 0.66$ for $\alpha = 0.2$). If β has its optimum value, then a combined sublinearity with exponent $2\sqrt{2-\alpha} + (2-\alpha)$ can be achieved, which for exact queries means $O(n^{0.83})$ space and search time. It is not possible to obtain a lower combined sublinearity.

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

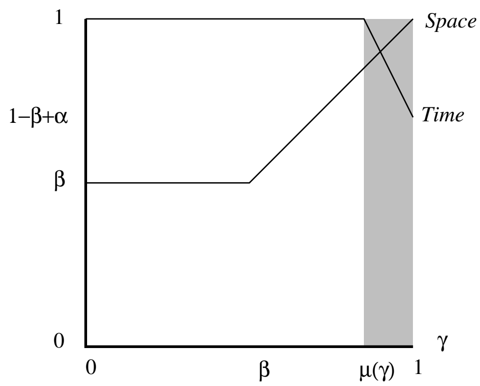

Figure 2: On the left, valid combinations for time and space complexity assuming $\theta = 1.87$, $\beta = 0.4$ and $\alpha = 0.18$. On the right, time and space complexity exponents. The area of combined sublinearity is shaded.

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. Splitting blocks due to deletions in the text collection is however more complicated, but many collections never decrease significantly in size.

5 Experimental Validation

In this section we validate experimentally the previous analysis. For our experiments, we make use of the TREC-2 collection [18]. 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). Table 1 presents some statistics about the five text files. We considered a word as a contiguous string of characters in the set {A...Z, a...z, 0...9} separated by other characters not in the set {A...Z, a...z, 0...9}. In the vocabulary the words are processed in a case-insensitive fashion.

We obtain our empirical values of β and θ from these files. In this section we use mainly the WSJ collection, which contains 250 Mb of text, 200 Mb of which is indexable (i.e. after remotion of stop-words and separators). Throughout this section 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.

Files	Text		Vocabulary		Vocab./Text		β	θ
	Size (bytes)	#Words	Size (bytes)	#Words	Size	#Words		
ap	266,533,400	41,849,991	1,785,467	193,550	0.67%	0.46%	0.46	1.87
doe	192,723,764	28,997,050	1,770,336	179,311	0.92%	0.62%	0.52	1.70
fr	272,323,115	39,560,980	1,053,680	117,964	0.39%	0.30%	0.48	1.94
wsj	279,534,695	43,392,799	1,379,793	159,727	0.49%	0.37%	0.40	1.87
ziff	253,177,255	39,426,198	1,404,570	161,502	0.55%	0.41%	0.51	1.79

Table 1: Text files from the TREC collection.

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.

5.1 Vocabulary Growth

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.

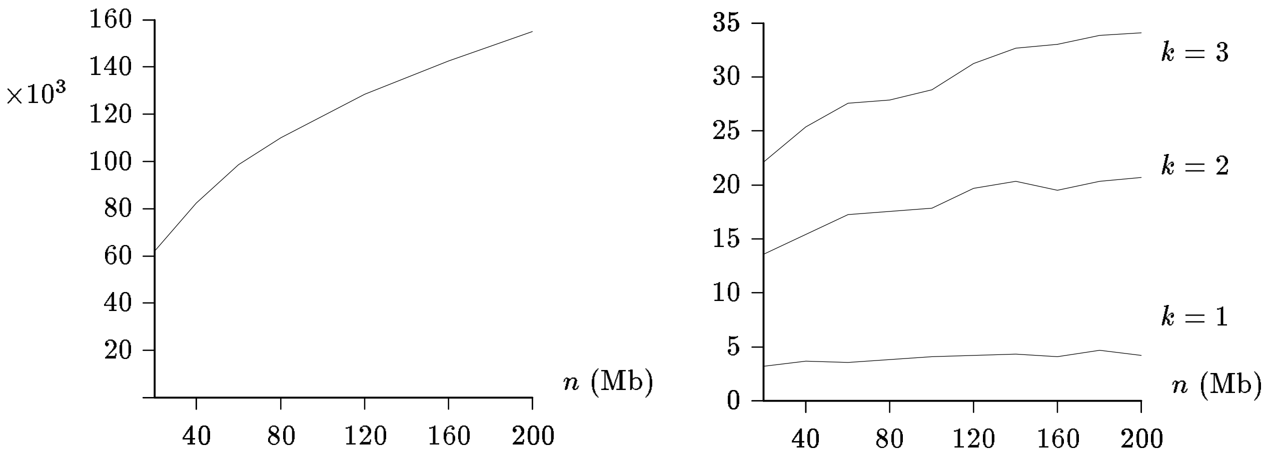

Figure 3: Vocabulary tests for the WSJ collection. On the left, the number of words in the vocabulary. On the right, number of vocabulary words matching a query.

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.

Observe that we could reduce the variance in the experiments by selecting once the set of queries from the index of the first 20 Mb. However, our experiments have shown that this is not a good policy. The reason is that the first 20 Mb will contain almost all common words, whose occurrence lists grow faster than the average. Most uncommon words will not be included. Therefore, the result is unfair, making the times to look linear when they are in fact sublinear.

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.

Figure 4 shows the evolution of the β value as the text collection grows. We show its value for up to 1 Mb. As it can be seen, β starts at a higher value and converges to the definitive 0.40 as the text grows. For 1 Mb it has almost reached its definitive value.

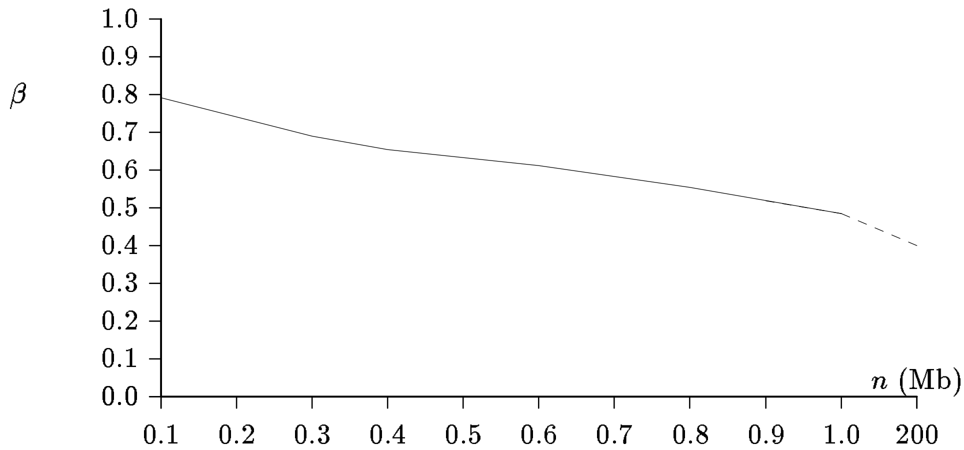

Figure 4: Value of β as the text grows. We added at the end the value for the 200 Mb collection.

5.2 Space versus Time for Fixed Block Size

We show the space overhead of the index and the time to answer queries allowing $k = 2$ errors, for three different fixed block sizes: 2 Kb, 32 Kb and 512 Kb. See Figure 5. 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, especially for $b = 2$ Kb. This happens 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.

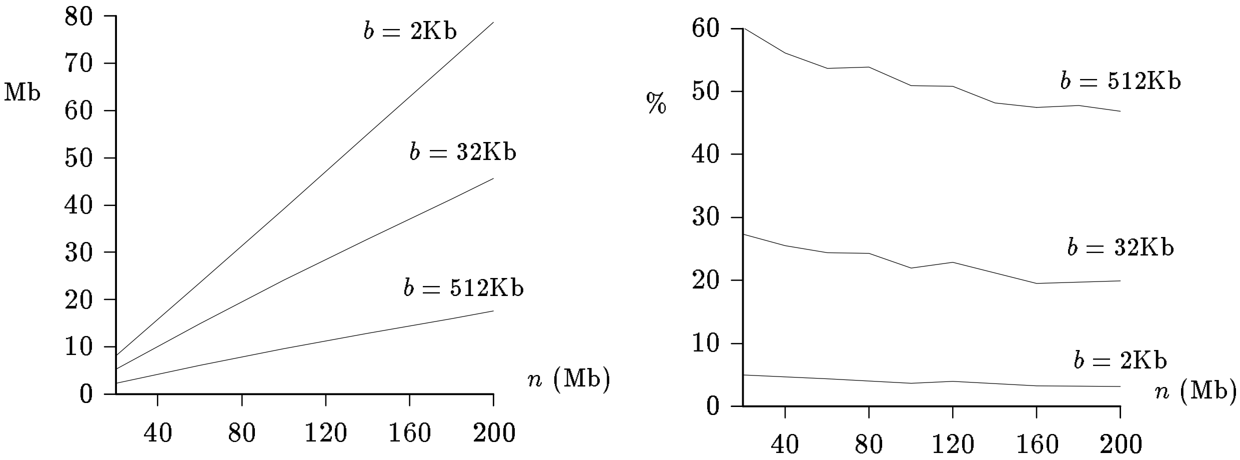

Figure 5: Experiments for fixed block size b. On the left, space taken by the indices. On the right, percentages of the text sizes sequentially searched allowing $k = 2$ errors.

5.3 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 6, where we measure again space overhead and query times allowing $k = 2$ errors, 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 curve for smaller r matches the analysis better, for similar reasons (i.e. the effect is noticed sooner for smaller r).

On the other hand, the percentage of the traversed text increases. This is because the proportion of text traversed (Eq. (1)) is $(1 - e^{-Theta(n^{1-\beta+\alpha})})$, which tends to 1 from below as n grows.

5.4 Sublinear Space and Time

Finally, we show experimentally in Figure 7 that time and space can be simultaneously sublinear. We test $\gamma = 0.92$, 0.94 and 0.96, again for $k = 2$ errors. 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: $0.40n^{0.89}$, $0.41n^{0.92}$ and $0.42n^{0.95}$. The error of the approximations is under 1%.

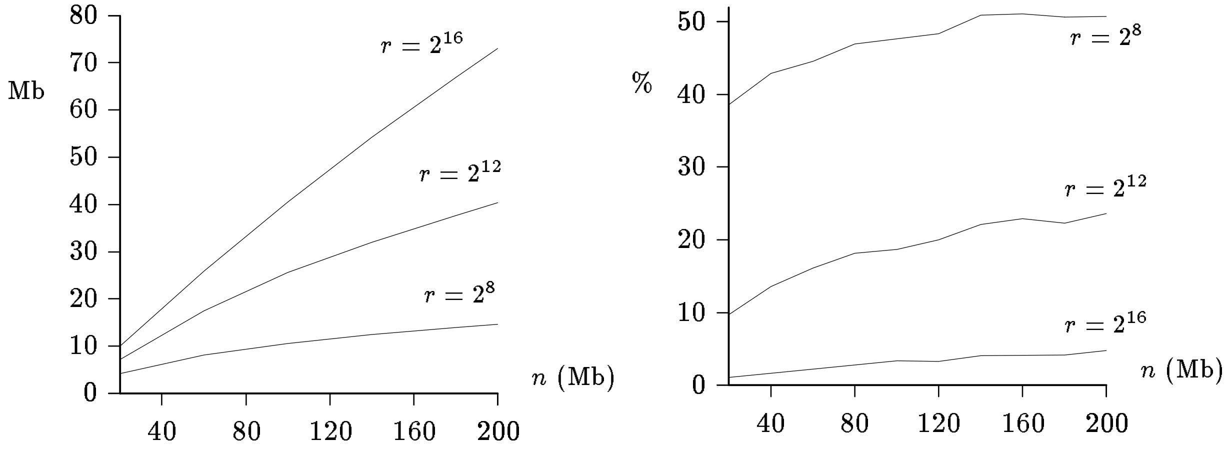

Figure 6: Experiments for fixed number of blocks r. On the left, space taken by the indices. On the right, percentages of the text sizes sequentially searched, allowing $k = 2$ errors.

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 an $O(n^{0.94})$ space and time index if 2 errors are allowed (our analysis predicts $O(n^{0.93})$).

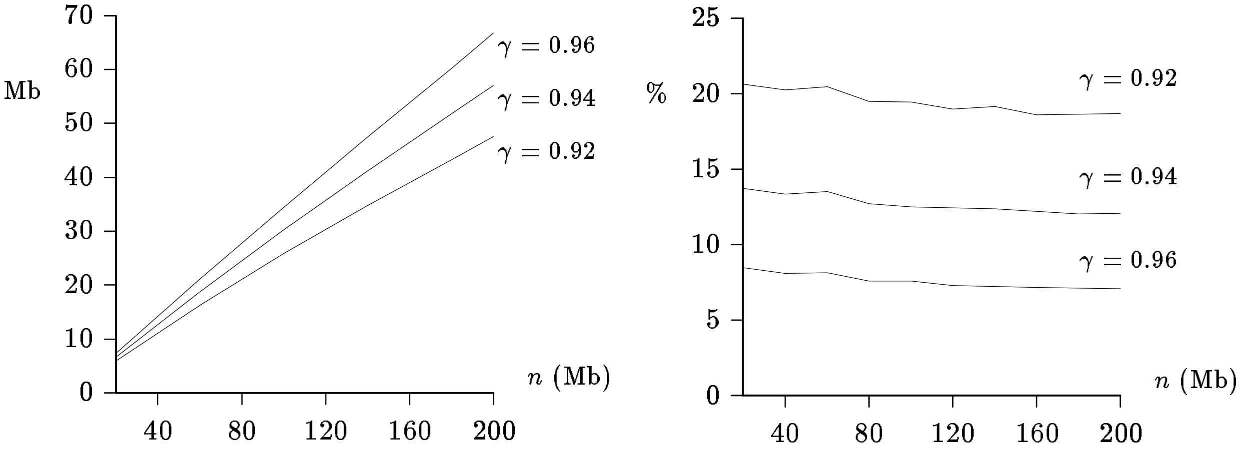

Figure 7: Experiments for fixed γ (simultaneous sublinearity). On the left, space taken by the indices. On the right, percentages of the text sizes sequentially searched, allowing $k = 2$ errors.

As another example, we show in Figure 8 the results on simultaneous sublinearity for the ZIFF collection, which has near 220 Mb after remotion of stop-words and separators. The values for this collection are $\beta = 0.51$ and $\theta = 1.79$. Least squares show a very good agreement with the analysis: we have $0.71n^{0.92}$ for $\gamma = 0.92$, $0.60n^{0.94}$ for $\gamma = 0.94$ and $0.55n^{0.95}$ for $\gamma = 0.96$. The relative error is below 0.5%. The times, for $k = 2$ errors, give $0.22n^{0.99}$ for $\gamma = 0.92$, $0.17n^{0.98}$ for $\gamma = 0.94$ and $0.14n^{0.96}$ for $\gamma = 0.96$. Hence, we can have an $O(n^{0.96})$ space and time index for ZIFF. It is interesting to notice that, although ZIFF has a larger vocabulary than WSJ, the results are not better. This is because the number of matching words in the vocabulary is also higher.

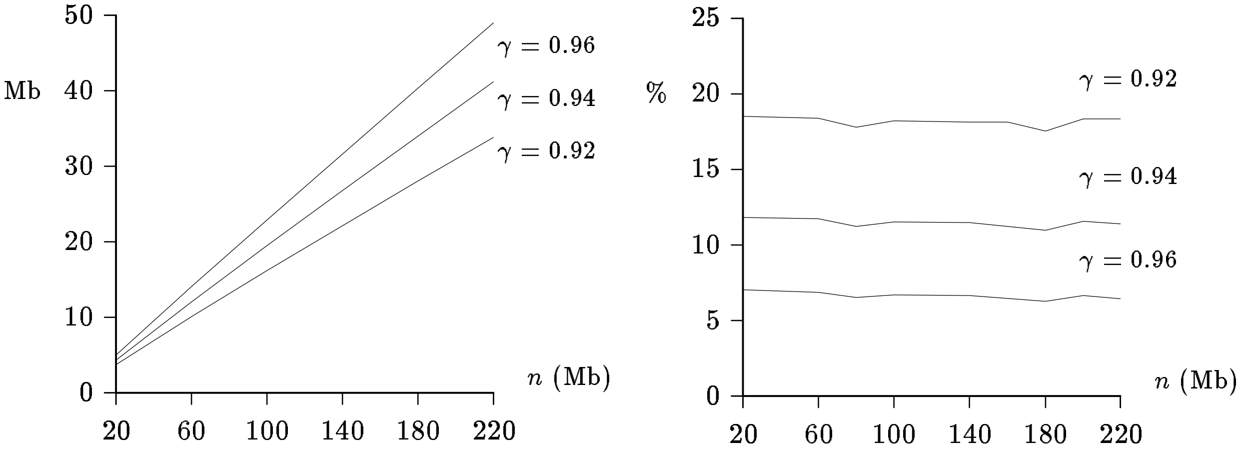

Figure 8: Experiments for fixed γ (simultaneous sublinearity) for the ZIFF collection. On the left, space taken by the indices. On the right, percentages of the text sizes sequentially searched, allowing $k = 2$ errors.

6 Application: Analyzing the Web

In [13], an empirical model for the distribution of the sizes of the Web pages is presented, backed by thorough experiments[4]. This distribution is as follows: the probability that a Web page is of size x is

$$p(x) \quad = \quad \frac{\lambda k^{\lambda}}{x^{1+\lambda}}$$

for $x \geq k$, and zero otherwise. The cummulative distribution is

$$F(x) \quad = \quad 1 - \left(\frac{k}{x}\right)^{\lambda}$$

where k and λ are constants dependent on the particular collection: k is the minimum document size, and $\lambda = 1.36$ when considering only textual data.

As explained before, pointing to documents instead of blocks may or may not be convenient in terms of query times. We analyze now the space and time requirements when we point to Web pages. We analyze space first.

As the Heaps' law states that a document with x words has x^{β} different words, we have that each new document of size x added to the collection will insert x^{β} new references to the lists of occurrences (since each different word of each different document has an entry in the index). Hence

[4] The model was refined in [6], but this does not change the asymptotic results we obtain here.

the average number of new entries in the occurrence list per document is

$$\int_k^\infty p(x) x^\beta dx \quad = \quad \frac{\lambda k^\beta}{\lambda - \beta} \tag{4}$$

To determine the total size of the collection, we consider that r documents exist, whose average length is

$$b^* \quad = \quad \int_k^\infty p(x) x dx \quad = \quad \frac{\lambda k}{\lambda - 1} \tag{5}$$

and therefore the total size of the collection is

$$n \quad = \quad \frac{r \lambda k}{\lambda - 1} \tag{6}$$

The size of the vocabulary in the final collection is

$$n^\beta \quad = \quad \left(\frac{r \lambda k}{\lambda - 1} \right)^\beta$$

and the final size of the occurrence lists is (using Eqs. (4) and then (6))

$$\frac{r \lambda k^\beta}{\lambda - \beta} \quad = \quad \frac{\lambda - 1}{\lambda - \beta} \, \frac{1}{k^{1-\beta}} \, n \tag{7}$$

A first result is that the space of the index is $\Theta(n)$ (this should be clear as $b^* = O(1)$). We consider now what happens if we take the average document length and use blocks of that fixed size (splitting long documents and putting short documents together as explained). In this case, the size of the vocabulary is $O(n^\beta)$ as before, and we assume that each block is of a fixed size $b = zb^* = z\lambda k/(\lambda - 1)$ (Eq. (5)). We have introduced a constant z to control the size of our blocks. In particular, if we use the same number of blocks as Web pages, then $z = 1$. Then the size of the lists of occurrences is

$$(r/z) b^\beta \quad = \quad r \frac{\lambda^\beta k^\beta z^\beta}{z(\lambda - 1)^\beta} \quad = \quad \left(\frac{\lambda - 1}{z \lambda k} \right)^{1-\beta} \quad n$$

(using Eq. (6) for the last step). Now, if we divide the space taken by the index of documents by the space taken by the index of blocks (using the previous equation and Eq. (7)), the ratio is

$$\frac{\text{doc. index size}}{\text{block index size}} \quad = \quad \frac{z^{1-\beta} \lambda^{1-\beta} (\lambda - 1)^\beta}{\lambda - \beta} \tag{8}$$

which is independent on k and rounds 80% for $z = 1$ and $\beta = 0.4..0.6$. This shows that indexing documents yields an index which takes 80% of the space of a block addressing index, if we have as many blocks as documents. Figure 9 shows the ratio as a function of λ and β. As it can be seen, the result varies slowly with β, while it depends more on λ (tending to 1 as the document size distribution is more uniform).

The fact that the ratio varies so slowly with β is good because we already know that the β value is quite different for small documents. As a curiosity, notice that there is a β value which gives the minimum ratio for document versus block index (i.e. the worst behavior for the block index). This is $\beta^* = \lambda - 1/\ln(z\lambda/(\lambda-1))$, which is $\beta^* \approx 0.61$ for $z = 1$.

If we want to have the same space overhead for the document and the block indices, we simply make the expression of Eq. (8) equal to 1 and obtain $z \approx 1.4..1.7$ for $\beta = 0.4..0.6$, i.e. we need to make the blocks larger than the average of the Web pages. This translates into worse search times. By paying more at search time we can obtain smaller indices (letting z grow over 1.7).

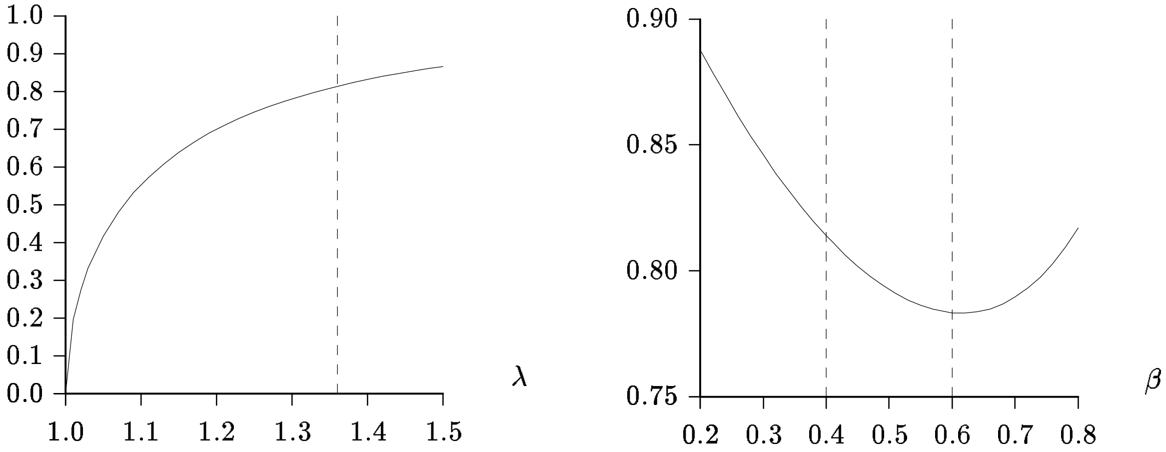

Figure 9: On the left, ratio between both indices as a function of λ for fixed $\beta = 0.5$ (the dashed line shows the actual λ value for the Web). On the right, the same as a function of β for $\lambda = 1.36$ (the dashed lines enclose the typical β values). In both cases we use the standard $z = 1$.

To show how retrieval times are affected by a non-uniform distribution when we have to traverse the matching blocks, we do the analysis for the document size distribution of the Web. As we have shown, if a block has size x then the probability that it has to be traversed is $(1 - e^{-\Theta(x/n^{\beta-\alpha})})$. We multiply this by the cost x to traverse it and integrate over all the possible sizes, so as to obtain its expected traversal cost (recall Eq. (1))

$$\int_k^\infty x(1 - e^{-\Theta(x/n^{\beta-\alpha})})p(x)dx$$

which we cannot solve. However, we can separate the integral in two parts. (a) $x = o(n^{\beta-\alpha})$ and (b) $x = \Omega(n^{\beta-\alpha})$. In the first case the traversal probability is $O(x/n^{\beta-\alpha})$ and in the second case it is $\Omega(1)$. Splitting the integral in two parts we obtain

$$\Theta\left(\frac{\lambda k^\lambda}{2-\lambda}\, n^{(\beta-\alpha)(1-\lambda)} + \frac{\lambda k^\lambda}{\lambda - 1}\, n^{(\beta-\alpha)(1-\lambda)}\right) = \Theta\left(\frac{\lambda k^\lambda}{(2-\lambda)(\lambda-1)}\, n^{(\beta-\alpha)(1-\lambda)}\right)$$

Now that we have the cost per block, we multiply by $r = (\lambda - 1)n/(\lambda k)$ (Eq. (6)) to obtain the total amount of work. This is

$$\frac{k^{\lambda-1}}{2-\lambda}\, n^{1-(\lambda-1)(\beta-\alpha)}$$

On the other hand, if we used blocks of fixed size, the time complexity (using Eq. (2)) would be $O(bn^{1-\beta+\alpha})$, where $b = zb^*$. The ratio between both search times is

$$\frac{\text{doc. index traversal}}{\text{block index traversal}} = \frac{(\lambda-1)n^{(2-\lambda)(\beta-\alpha)}}{\lambda(2-\lambda)zk^{2-\lambda}}$$

which shows that the document index would be asymptotically slower than a block index as the text collection grows. In practice, the ratio is between $O(n^{0.2})$ and $O(n^{0.4})$. The value of z is not important here since it is a constant.

7 Conclusions and Future Work

We focused on the problem of block addressing for approximate word retrieving indices. These indices address two problems of outmost importance in modern textual databases: space overhead and querying flexibility.

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), even when errors are allowed in the match. We also 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 queries in $O(n^{0.94})$ time allowing 2 errors. Those results apply to classical queries too, not only to approximate searching. For instance, for the same text collection it would be possible to answer classical queries in $O(n^{0.85})$ time and space. We finally analyze the query times and space requirements of an index which points to Web pages, considering their typical distribution.

It would be interesting to investigate this tradeoff in other contexts. For instance, approximate queries on patterns of length m can be answered using a suffix tree [1] of the text in time exponential with m but independent on the text size n. The suffix tree takes linear space (with a constant factor that makes it impractical for information retrieval applications but still attractive for computational biology). On the other hand, the problem can be solved in $O(n)$ time with sequential searching, using an automaton whose size is exponential in m [35]. It is not known whether it is possible to obtain a tradeoff between these worst-case complexities.

There are also interesting issues for future research more on the practical side. The sequential search on the blocks, if needed, dominates the time to search in the vocabulary. We have presented in [3] an approach based on replacing the approximate searching on the blocks by a multipattern searching (since after the vocabulary search we know exactly which words match the query). This idea is in fact the essence of the success of *Cgrep* [29].

On the other hand, when the sequential search is not necessary, improving the vocabulary search is of interest. In [4] we have shown how approximate searching can be improved by structuring the vocabulary as a metric space. This relies on the fact that the edit distance is a metric.

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 us 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 significant in practice (up to 50%). We are also studying compression schemes to search directly in the compressed text [29, 28], such that not only the index points to blocks to reduce its space, but it is compressed. The text is also compressed and the sequential search inside the compressed text blocks is efficiently done using *Cgrep*.

References

[1] A. Apostolico. The myriad virtues of subword trees. In *Combinatorial Algorithms on Words*, NATO ISI Series, pages 85–96. Springer-Verlag, 1985.

[2] M. Araújo, G. Navarro, and N. Ziviani. Large text searching allowing errors. In *Proc. WSP'97*, pages 2–20. Carleton University Press, 1997.

[3] R. Baeza-Yates and G. Navarro. Block-addressing indices for approximate text retrieval. In *Proc. ACM CIKM'97*, pages 1–8, 1997.

[4] R. Baeza-Yates and G. Navarro. Fast approximate string matching in a dictionary. In *Proc. SPIRE'98*, pages 14–22. IEEE Computer Press, 1998.

[5] R. Baeza-Yates and B. Ribeiro. *Modern Information Retrieval.* Addison-Wesley, 1999.

[6] P. Barford and M. Crovella. Generating representative Web workloads for network and server performance evaluation. In *Proc. ACM SIGMETRICS 1998*, pages 151–160, 1998.

[7] T. Bell, J. Cleary, and I. Witten. *Text Compression.* Prentice Hall, New Jersey, 1990.

[8] T. Bell, A. Moffat, C. Nevill-Manning, I. Witten, and J. Zobel. Data compression in full-text retrieval systems. *J. of the American Society for Information Science*, 44:508–531, 1993.

[9] T. Bell, A. Moffat, I. Witten, and J. Zobel. The MG retrieval system: compressing for space and speed. *CACM*, 38(4):41–42, 1995.

[10] C. Blair. A program for correcting spelling errors. *Information and Control*, 3:60–67, 1960.

[11] K. Church and W. Gale. Poisson mixtures. *Natural Language Engineering*, 1(2):163–190, 1995.

[12] A. Cobbs. Fast approximate matching using suffix trees. In *Proc. CPM'95*, pages 41–54, 1995.

[13] M. Crovella and A. Bestavros. Self-similarity in World Wide Web: evidence and possible causes. In *Proc. ACM SIGMETRICS 1996*, pages 160–169, 1996.

[14] F. Damerau. A technique for computer detection and correction of spelling errors. *Comm. of the ACM*, 7(3):171–176, 1964.

[15] W. Frakes and R. Baeza-Yates, editors. *Information Retrieval: Data Structures and Algorithms.* Prentice-Hall, 1992.

[16] G. Gonnet. A tutorial introduction to Computational Biochemistry using Darwin. Technical report, Informatik E.T.H., Zurich, Switzerland, 1992.

[17] P. Hall and G. Dowling. Approximate string matching. *ACM Computing Surveys*, 12(4):381–402, 1980.

[18] 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.

[19] H.S. Heaps. *Information Retrieval - Computational and Theoretical Aspects.* Academic Press, NY, 1978.

[20] D. Huffman. A method for the construction of minimum-redundancy codes. *Proc. of the I.R.E.*, 40(9):1090–1101, 1952.

[21] D. Knuth. *The Art of Computer Programming*, volume 3: Sorting and Searching. Addison-Wesley, 1973.

[22] O. Lehtinen, E. Sutinen, and J. Tarhio. Experiments on block indexing. In *Proc. WSP'96*, pages 183–193. Carleton University Press, 1996.

[23] V. Levenshtein. Binary codes capable of correcting spurious insertions and deletions of ones. *Problems of Information Transmission*, 1:8–17, 1965.

[24] V. Levenshtein. Binary codes capable of correcting deletions, insertions and reversals. *Soviet Physics Doklady*, 10(8):707–710, 1966. Original in Russian in *Doklady Akademii Nauk SSSR, 163(4):845–848, 1965.*

[25] 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.

[26] H. Masters. A study of spelling errors. *Univ. of Iowa Studies in Education*, 4(4), 1927.

[27] A. Moffat. Word-based text compression. *Software Practice and Experience*, 19(2):185–198, 1989.

[28] E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Direct pattern matching on compressed text. In *Proc. of the 5th South American Symposium on String Processing and Information Retrieval (SPIRE'98)*, pages 90–95. IEEE CS Press, 1998. `ftp://ftp.dcc.uchile.cl/pub/-users/gnavarro/spire98.3.ps.gz`.

[29] E. Moura, G. Navarro, N. Ziviani, and R. Baeza-Yates. Fast searching on compressed text allowing errors. In *Proc. SIGIR'98*, pages 298–306. York Press, 1998.

[30] E. Myers. A sublinear algorithm for approximate keyword searching. *Algorithmica*, 12(4/5):345–374, Oct/Nov 1994.

[31] G. Navarro. *Approximate Text Searching.* PhD thesis, Dept. of Computer Science, Univ. of Chile, December 1998. Technical Report TR/DCC-98-14. `ftp://ftp.dcc.uchile.cl/pub/-users/gnavarro/thesis98.ps.gz`.

[32] G. Navarro and R. Baeza-Yates. A new indexing method for approximate string matching. Technical Report TR/DCC-98-14, Dept. of Computer Science, Univ. of Chile, December 1998. `ftp://ftp.dcc.uchile.cl/pub/users/gnavarro/stindex.ps.gz`.

[33] G. Navarro and R. Baeza-Yates. A practical q-gram index for text retrieval allowing errors. *CLEI Electronic Journal*, 1(2), 1998. `http://www.clei.cl`.

[34] J. Nesbit. The accuracy of approximate string matching algorithms. *J. of Computer-Based Instruction*, 13(3):80–83, 1986.

[35] E. Ukkonen. Finding approximate patterns in strings. *J. of Algorithms*, 6:132–137, 1985.

[36] I. Witten, A. Moffat, and T. Bell. *Managing Gigabytes.* Van Nostrand Reinhold, New York, 1994.

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

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

Appendix: Space Analysis Using Zipf's Law

We analyze the space usage using Zipf's law instead of Heaps' law, as is done in the paper. The analysis is more complex in this case.

Suppose that a word appears ℓ times in the text. The same argument used for Eq. (1) shows that it appears in $\Theta(r(1-e^{-\Theta(\ell/r)}))$ 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

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

and the total number of references to blocks is

$$r \sum_{i=1}^{V} 1 - e^{-\Theta(b/(i^\theta H_V^{(\theta)}))} \tag{9}$$

a summation which is hard to solve exactly. However, we can still obtain the required big-O information. We show that there is a threshold

$$a = \left(\frac{b}{H_V^{(\theta)}}\right)^{1/\theta}, \quad \text{such that}$$

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 (9) 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 $r(1-e^{-\Theta(\ell/r)}) = \Theta(\ell)$ 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 consider the case $\theta = 1$ shortly).

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

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. 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.