Searching Large Lexicons for Partially Specified Terms using Compressed Inverted Files

Justin Zobel* Alistair Moffat† Ron Sacks-Davis‡

Abstract

There are many advantages to be gained by storing the lexicon of a full text database in main memory. In this paper we describe how to use a compressed inverted file index to search such a lexicon for entries that match a pattern or partially specified term. This method provides an effective compromise between speed and space, running orders of magnitude faster than brute force search, but requiring less memory than other pattern-matching data structures; indeed, in some cases requiring less memory than would be consumed by a single pointer to each string. The pattern search method is based on text indexing techniques and is a successful adaptation of inverted files to main memory databases.

1 Introduction

Given the large main memories available on current computers, it is interesting to ask what additional facilities might be incorporated in full text retrieval systems if memory usage is allowed to expand. One possible application for extra memory is to store a comprehensive compression model for the text of the document collection, thereby substantially increasing the effective capacity of the storage devices being used [3]. Another possibility is to store the lexicon of the collection in main memory, eliminating the disk accesses needed to search for query terms and improving query performance [24]. Indeed, these two functions might be accomplished using the same structure [15, 23].

Here we examine a further facility that can be provided if the lexicon is held in memory. The lexicon, which is part of the index, is a structure containing all of the words occurring in the database. Queries, which are used to find documents containing certain words, are evaluated by finding the words in the lexicon, then retrieving and merging the postings lists of the words. Given this organisation, it is natural to consider methods by which an in-memory lexicon can be used to support expansion of partially specified query terms, that is, to support queries on incomplete words or word fragments.

Standard languages for interactive text retrieval include pattern-matching constructs such as wild characters and other forms of partial specification of query terms [12]. Certainly, if the lexicon is available in main memory it can be scanned using normal pattern matching techniques to locate partially specified terms. For all but very small lexicons, however, linear search is prohibitively expensive, and is not a viable option.

We describe a solution that is both fast in operation and economical of additional storage space. A lexicon should be treated as a database that can be accessed via an index of fixed-length substrings, or *n-grams*, of the words in the lexicon. To retrieve strings that match a pattern, all of the n-grams in the pattern are extracted; the words in the lexicon that contain these substrings are identified via the index; and these words are checked against the pattern for *false matches*. Using this technique, index size can be traded against retrieval time by varying n. However, straightforward application of n-gram indexing techniques—well known in information retrieval folklore as a method of resolving partially specified query terms—does not give particularly good

*CITRI, Department of Computer Science, RMIT, GPO Box 2476V, Melbourne 3001, Australia; jz@cs.rmit.oz.au

†Department of Computer Science, The University of Melbourne, Parkville 3052, Australia; alistair@cs.mu.oz.au

‡Collaborative Information Technology Research Institute, 723 Swanston St., Carlton 3052, Australia; rsd@kbs.citri.edu.au

Proceedings of the 19th VLDB Conference Dublin, Ireland, 1993

performance, and fast retrieval is only achieved by use of indexes that are much larger than the lexicon itself.

In this paper we show how to improve performance by use of several optimisations. First, we describe an inverted file indexing technique that employs fast compression to reduce space requirements. Second, since false match checking is very fast in this application, we reduce time by only checking for some n-grams. Third, both index size and query time can be further reduced by considering short blocks of words as a single string. Fourth, in those applications in which sorting or partial sorting of the lexicon is possible, additional savings in both size and time can be achieved. Our experimental results demonstrate the effect of each of these optimisations, and show that they make possible fast access to even large lexicons such as that of a multi-gigabyte text database containing over 800,000 terms. In contrast to our techniques, previous methods are either very slow or require large space overheads.

This paper is organised as follows. In the following section we discuss our test data and the mechanism used to evaluate different lexicon search schemes. In Section 3 we discuss previous proposals for searching lexicons. We provide a description of inverted file indexing techniques in Section 4. Our lexicon indexing technique is described, with detailed experimental results, in Section 5. Conclusions are presented in Section 6.

2 Test data

We have used three lexicons in our experiments: *Bible*, the complete lexicon of the 4.4 Mb King James version of the Bible, with original case preserved; *Macq*, the set of lowercase terms used in the 1990 edition of the Macquarie Encyclopedic Thesaurus; and *TREC*, the lexicon of 742,985 articles totalling 2 Gb extracted from the TIPSTER collection [18], again with original case preserved. The parameters of these lexicons are shown in Table 1. The 'Size' figures in the first row include all of the characters of the words comprising the lexicon, plus one overhead byte per word for termination. No indexing structures of any kind—not even an array of string pointers to the terms themselves—are included. Throughout this paper we express all index sizes as a percentage relative to the raw cost of these strings. For example, if we decided to index each string in the *TREC* lexicon with a four byte pointer, we would allow $4 \times 805,590$ bytes and describe the index as occupying an overhead of 42.8%; similarly, a 600 Kb index for the 537.3 Kb *Macq* lexicon would have size 111.7%.

	Bible	*Macq*	*TREC*
Size (Kb)	107.1	537.3	7,348.3
Number of words	13,688	58.164	805,590
Av. word length (chars)	7.02	8.46	8.34
Size of alphabet	51[†]	26	52

Table 1: Parameters of lexicons

To test the effectiveness of different techniques for matching a pattern against a lexicon, a query set is needed. We chose to generate a set of queries by taking a random selection of 250 words defined in the first edition of Longman's Dictionary of Contemporary English. We then used this query set in two ways: as *full*, or fully-specified queries (that is, without any wildcards); and as *part*, or partially-specified queries, by transforming them into patterns that included wildcards. This was done by randomly replacing substrings of these words by `*`, where the substrings were of one or more characters and `*` represents zero or more occurrences of *any* character. In this transformation, we ensured that the resulting pattern contained either a leading or trailing string of length three, or an embedded string of length four; we believe that the resulting patterns were representative of patterns that might be used in practice. For example, `frozen` might be transformed to `fro*n`, which will match (among other things) `frogspawn`, `frogman`, and `frown`. Note that the pattern must match the whole string, not a subpart of it; for example. to fetch strings containing substrings that match `fro*n`, such as `leapfrogging`, the pattern `*fro*n*` would be used.

Most of the words in the query set contained only lowercase letters, but a few proper nouns and acronyms containing uppercase letters were also (by luck) selected. Some strings from the query set, after transformation into patterns, are shown in Table 2; the reader may care to guess the word that each string is derived from. For the *part* query set, there were an average of 1.6 answers per query in *Bible*, 14.3 answers per query in *Macq*, and 68.2 answers per query in *TREC*; for the *full* query set, the figures were 0.14, 0.84, and 0.89 respectively.

In pattern matching, features other than 'match any substring' can be used, such as searching for strings with repeated patterns, or for strings with any of a given set of characters in a certain position. We do not demon-

[†]Curiously, the King James version of the Bible doesn't use the uppercase letter 'X'.

*gger	a*t*labe	*n*oke
bu*toot*	pushcar*	tel*metry
departmen*	a*oured	lam*ik*
min*cu*e	*n*ene	*it*og*ycerin*
amen*s	*u*west*n	witch*aft
crene*la*ed	*a*e*ous	t*opic
classle*s	nut*h*ll	*rebook
*eticence	dis*rimin*	fruc*fication

Table 2: Example query strings

strate our method with such patterns, but do indicate how they would be supported.

3 Previous lexicon search techniques

We assume for generality that a lexicon is a series of strings, in no particular order, separated by single-byte terminators. In practice, other information such as occurrence counts and disk pointers will also be stored, but we assume that these are stored in parallel arrays indexed by ordinal string number; we do not consider those additional structures in this paper. Some lexicon searching techniques also require that some or all of the strings be indexed with a pointer. Where this is necessary we have counted the space required for this as part of the index, allowing $\lceil \log_2 C \rceil$ bits for each pointer, where C is the total number of characters in the lexicon.

The simplest way to find the strings in a lexicon that match a given pattern is to use brute force, that is, search the lexicon from beginning to end. This requires only that it be possible to traverse the lexicon in some orderly fashion: the lexicon does not have to be sorted, and no additional structures are needed.

To effect the pattern matching itself, finite automata techniques [1] such as the UNIX *regex* package can be used. This package provides reasonably fast pattern matching over a rich pattern language. (There are algorithms that are in general much faster than the *regex* package, but for short patterns and strings these algorithms may not be significantly better, because their greater initial overheads can dominate the cost.) To match a pattern against a series of strings, a function is used to compile the pattern into an internal form, and then, for each string, another function is used to check whether the pattern and string match.

To test the speed of brute force search on a lexicon, we found each string in the lexicon by searching for the end-of-string delimiters—recall that we have not, as yet, allowed any string pointers—and used *regex* to check whether the string matched the query pattern. In the first section of Table 3 we show the average speed per query of brute force search on our lexicons. The evaluation times, as for all the evaluation times given in this paper, are average cpu milliseconds per query on a 25 MIP Sun SPARC 2, and do not include any input or output operations other than the reading of the queries themselves.

As can be seen, *regex* matches patterns at around 10 microseconds per word on average, or about one second of processing time for a lexicon of 100,000 words. This approach represents one extreme in the space-time tradeoff, at which the space overheads are small and independent of the size of the lexicon, but searching is slow. For reasonable response, more structured access is needed.

Bratley & Choueka [4] have proposed a *permuted dictionary* mechanism for processing partially specified terms in database queries. Their method uses a permuted lexicon that consists of all possible rotations of each word in the lexicon, so that, for example, the word **frozen** would contribute the original form **|frozen**, and the rotated forms **frozen|**, **rozen|f**, **ozen|fr**, **zen|fro**, **en|froz**, and **n|froze**, where the symbol **|** indicates the beginning of the word. The resulting set of strings is then lexicographically ordered. Using this mechanism, all patterns of the form X^*, $^*X$, $^*X^*$, and X^*Y can be processed by binary search in the permuted lexicon. For example, the pattern **fro*n** would be rotated to generate **n|fro***. A search in the sorted lexicon for this prefix will return, amongst others, the entry **n|froze**, from which the word **frozen** can be recovered.

Bratley & Choueka discuss prefix-omission mechanisms by which the space requirements of the permuted lexicon can be reduced. Even when compression is used the biggest drawback of this method is, nonetheless, the space requirement, since a word of n characters contributes $n+1$ entries to the permuted lexicon. For the 7,348 Kb *TREC* lexicon, for example, the permuted lexicon is 84,948 Kb, reducing to a minimum of 41,342 Kb after prefix-omission.

Following the approach of Gonnet & Baeza-Yates [11], we note that it is more effective to use a variant of this method in which the permuted lexicon is an array of pointers, one to each character position in the original lexicon. This array of pointers is sorted on the permuted form of each word; thus a pointer to the character **e** in **frozen** would be sorted on the value **en|froz**. The complete permuted form is identified by

	Bible		*Macq*		*TREC*	
Brute force	*part*	*full*	*part*	*full*	*part*	*full*
Evaluation time (ms)	144.5	100.3	665.5	441.5	10,228.9	7,513.2
Permuted lexicons						
Size of access structure (%)	212.6		250.0		287.5	
Creation time (sec)	25.7		157.9		2,970.1	
	part	*full*	*part*	*full*	*part*	*full*
Evaluation time (ms)	0.27	0.14	0.95	0.21	3.81	0.33
Pointer array						
Size of access structure (%)	26.5		26.4		30.8	

Table 3: Parameters of previous search techniques

searching backwards (when a string terminator is encountered) to locate the previous string terminator and thus the start of the current word; if additional structure is interleaved with the lexicon, each string may have to include an explicit code to allow the start of the current word to be identified. Space requirements for such permuted lexicons are shown in the second section of Table 3, assuming a $\lceil \log_2 C \rceil$-bit pointer to each of the C lexicon characters. (Recall that index size is expressed as a percentage of the size of the corresponding lexicon.) As can be seen, the search structures require roughly two to three times the space of the lexicons they index, and are very large.

To query the variant of the permuted lexicon method to find matches to a partially specified term, we used binary search on the pointer array to find, for each fragment of the term, the range of permuted strings that contained the fragment. We then used *regex* to select, from the smallest range of permuted strings, the strings that matched the query. For example, in the query **min*cu*e** there are two fragments, **cu** and **e|min**. In the permuted *Macq* lexicon there are 1,171 permuted strings that start with **cu** (from **cu|** to **cuzzi|ja**) and 14 strings that start with **e|min** (from **e|min** to **e|minutia**); to find the matches to **min*cu*e**, the 14 strings in the shorter range are checked with *regex*.

Evaluation times for the variant of the permuted lexicon method are shown in the second section of Table 3; as can be seen, queries are answered in milliseconds rather than seconds. This scheme is therefore at the other extreme to brute force search, with rapid lookup but substantial space overheads.

Patricia tries based on the semi-infinite strings in the lexicon are a generalisation of this approach [11, 13, 17]. In these methods the sequence of words is considered to be a contiguous string, and the trie used to determine the position in the string at which a given substring occurs. Although more general, these search methods are unlikely to be faster than the permuted dictionary, since they involve navigation of a lexicographic search tree. Moreover, they will, as a minimum, require at least one pointer for every character in the lexicon, and so will consume at least as much space as the pointer-based permuted lexicon described above.

Owolabi & McGregor have proposed a string search mechanism which, like the system we describe, uses an n-gram index to locate matching strings [19]. Their index is a form of signature file, with a bitmap in which columns correspond to n-grams and rows to lexicon entries. We discuss the performance of this mechanism later.

For comparison, the third section of Table 3 lists, as a percentage of the size of the lexicon, the storage required if each word (rather than each character) in the lexicon is indexed with a $\lceil \log_2 C \rceil$-bit pointer. A (sorted) pointer array would be a suitable search structure if all query terms were fully specified and if it could be binary searched. We shall return to this possibility in Section 5.4, when we discuss sorted lexicons.

4 Inverted file text indexing

In this section we describe inverted file text indexing techniques.

An index is a structure that is used to map from queryable entities to indexed items. For example, in a database system an index is used to map from entities such as names and bank account numbers to records containing data about those entities. A general inverted file index consists of two parts: a set of *inverted file entries* (sometimes known as *postings lists*), being lists of ordinal item numbers of the items containing each queryable entity; and a *search structure* for mapping from an entity to the location of its inverted file entry. In a text database system, the search structure would typically be a sorted array or search tree, a hash table, or, on secondary storage, a B-tree.

To map ordinal item numbers to addresses there must also be an *address table*. For a main memory database, the address table is a list of pointers to items. Candidate answers to conjunctive queries are found by finding and merging the inverted file entries of the entities specified in the query and then using the address table to locate the answers to the query. This merge takes the intersection (rather than the union) of the numbers in the inverted file entry, so that the result of the merge will be the numbers of the items containing all of the entities in the query. In some queries, retrieval and merging of a series of inverted file entries may leave so few candidate answers that it is cheaper to examine those items and check for false matches than to merge the remaining inverted file entries, particularly if the remaining entries are long. We shall exploit this possibility below.

One problem with inverted files is that uncompressed they can consume a great deal of space, potentially several times as much as the data they index. For this reason, compression of inverted file entries, or equivalently bitmaps, has been analysed by many authors, include Fraenkel & Klein [8] and Bookstein & Klein [2]. Our presentation is based on that of Moffat & Zobel [16], who compare a variety of bitmap compression techniques. In all of these schemes decompression is fast—about 50–100 Kb of compressed data can be decompressed in a second on the Sun SPARC 2.

Rather than compressing the series of item numbers in an inverted file entry, it is convenient to compress their *run length encoding*, that is, the series of differences between successive numbers [9, 10]. For example, the inverted file entry $(4, 5, 9, 11, 12, 17, \ldots)$ has the run length encoding $(4, 1, 4, 2, 1, 5, \ldots)$. This does not in itself yield any compression, but does expose patterns that can be exploited for compression purposes.

A simple run length compression method is to use the codes for integers described by Elias [6]. His γ code represents integer x as $\lfloor \log_2 x \rfloor + 1$ in unary (that is, $\lfloor \log_2 x \rfloor$ 0-bits followed by a 1-bit) followed by $x - 2^{\lfloor \log_2 x \rfloor}$ in binary (that is, x less its most significant bit); the δ code uses γ to code $\lfloor \log_2 x \rfloor + 1$, followed by the same suffix. The δ code is longer than the γ code for some values of x smaller than 15, but thereafter δ is never worse.

The γ and δ codes are instances of a more general coding paradigm as follows [8]. Let V be a (possibly infinite) vector of positive integers v_i, $i \geq 1$, where $\sum v_i \geq N$, the number of items being indexed. To code integer $x \geq 1$ relative to V we find k such that

$$\sum_{j=1}^{k-1} v_j < x \leq \sum_{j=1}^{k} v_j$$

and code k in some representation followed by the difference

$$d = x - \sum_{j=1}^{k-1} v_j - 1$$

in binary, using either $\lfloor \log_2 v_k \rfloor$ bits if $d < 2^{\lceil \log_2 v_k \rceil} - v_k$ or $\lceil \log_2 v_k \rceil$ bits otherwise. For example, γ is an encoding relative to the vector $(1, 2, 4, 8, 16, \ldots)$ with k coded in unary.

Consider another example. Suppose that the coding vector is (for some reason) chosen to be $(9, 27, 81, \ldots)$. Then if k is coded in unary, the values 1 through to 7 would have codes 1000 through to 1110, with 8 and 9 as 11110 and 11111 respectively, where in each case the leading 1 is the code for k and the remainder is the code for d. Similarly, run lengths of 10 through to $36 = 9 + 27$ would be assigned codes with a 01 prefix and either a 4-bit or a 5-bit suffix: 0000 for 10 through to 0100 for 14, then 01010 for 15 through to 11111 for 36.

The effectiveness of compression for an inverted file entry will vary with the choice of vector. One scheme, due to Teuhola [21], is to use the vector $V_T = (b, 2b, 4b, 8b, 16b, \ldots)$, where each entry has an associated b value. An appropriate choice of b is the median run length in the entry [16], again with k coded in unary. This scheme gives good compression because it exploits clustering, a phenomenon that is particularly likely to happen should the strings in the lexicon be sorted, since the same substrings will occur in many consecutive strings. Another scheme is to use the vector $V_G = (b, b, b, b, b, \ldots)$, where $b = 0.69N/p$ and p is the number of run lengths in the inverted file entry [15]. (There is no requirement for the values v_i to be distinct.) This scheme gives good compression when the indexed data is randomly distributed, and thus is good for unsorted lexicons. Results of application of these schemes to lexicon indexing are shown in Section 5.

None of these compression schemes use arithmetic coding or adaptive modelling, neither of which are effective in this application—arithmetic coding should be avoided because it requires significant computational resources, and adaptive modelling is not viable because it requires long runs of data to be effective. Note also that in all of the schemes mentioned, entries are compressed individually and can be efficiently created on the fly, and it is never necessary to rebuild the entire index after update.

5 Lexicon indexes

We propose that a lexicon index be used to find those strings in a lexicon that match a given pattern. In this index, the queryable entities are *n-grams*, that is, all n character substrings of the words in the lexicon. The concept of n-grams has been attributed to Shannon [20], and can also be used for tasks such as string distance measurement [22].

Given an inverted file of n-grams, it is straightforward to retrieve strings that match a pattern. First, all of the n-grams in the pattern must be extracted. Then the inverted file entries are found by looking up the n-grams in a search structure that contains, for each n-gram, the characters comprising that n-gram and the address in memory of the corresponding compressed inverted file entry. Next, the inverted file entries for those n-grams must be decompressed and merged, to identify the ordinal numbers of the strings that contain all of those n-grams. Last, the strings corresponding to those numbers must be accessed via the address table, and a pattern matcher such as *regex* used to eliminate false matches, which can occur even in fully specified patterns.

As an example of how n-grams can be used for indexing, consider 2-grams. The string **tense** contains the 2-grams **te**, **en**, **ns**, and **se**. The pattern **ten*** contains the 2-grams **te** and **en**, and the set of strings containing both of these 2-grams will include all strings beginning with **ten**. This set would also include the false match **enter**, which contains both **te** and **en** but does not contain **ten**; this false match would, however, be eliminated by *regex*. Figure 1 shows this arrangement of n-gram lookup table, inverted file entries, string pointers, and lexicon, all stored in memory.

The number of false matches can be reduced if substrings that start or end a string are marked as such. For example, using | to mark the start (and end) of strings, **tense** would have the additional 2-grams |t and e|, and **ten*** would have the additional 2-gram |t. The set of strings containing all three of the 2-grams of **ten*** does not include **enter**, eliminating that particular false match. In the special case of n-grams of length 1, this technique of marking start and end symbols could not be used. An alternative measure is to extend the alphabet to distinguish between occurrences of the same character at the beginning, middle, or end of a string.

There is a trade-off to be made in choosing n. For large n, such as $n = 4$, there will be a large number of distinct n-grams, and false matches will be rare. Few strings, for example, would contain both **bein** and **eing** but not **being**—there were no such strings in any of the test vocabularies.

Since each 4-gram has few occurrences, the decompression and processing of index entries can be fast. Some query patterns will not contain any 4-grams (although we believe such patterns will be rare), in which case there is no option but to use brute force search. Also, as our results show, large n implies a large index. On the other hand, a small value of n, such as $n = 1$, not only leads to a large number of false matches, but also leads to high index processing costs. In Table 4 we show, for each n from 1 to 5, the number of distinct n-grams and the average number of occurrences of each n-gram in each of our test lexicons.

	n	*Bible*	*Macq*	*TREC*
Number of distinct n-grams	1	51	26	52
	2	920	620	2,778
	3	6,200	6,993	62,647
	4	19,135	37,541	372,849
	5	30,282	90,619	1,003,306
Average number of occurrences of each n-gram	1	1,882.9	18,922.6	129,212.0
	2	119.3	887.3	2,998.6
	3	15.5	70.4	120.1
	4	4.3	11.6	18.0
	5	2.7	4.8	5.9

Table 4: Numbers of n-grams

In the remainder of this section we show how n-grams can be used in conjunction with compressed inverted files. In our experiments we have only considered n of 2, 3, and 4; preliminary investigations showed that 1-grams were unacceptably slow—as would be expected, given the length of each inverted file entry and the number of false matches to be eliminated—and that the space requirements for 5-grams were unacceptably high.

Note that n-grams can be used to support other kinds of pattern matching. For example, if $n = 3$ and patterns contain sequences such as **ab[cd]e**, where the square brackets denote that the character between **b** and **e** must be either **c** or **d**, then matches can be found by looking for strings containing either **abc** and **bce** or **abd** and **bde**.

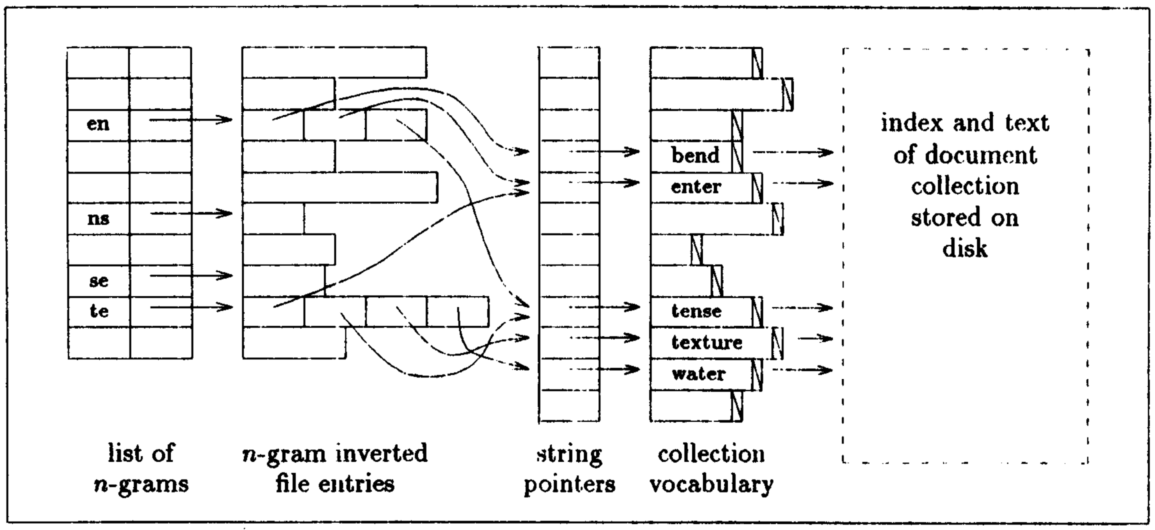

Figure 1: In-memory n-gram index for vocabulary strings

5.1 Naive inverted file n-gram indexing

We first investigate indexes for unsorted lexicons, in which words appeared in order of first occurrence in the source collection, using as the only 'improvement' the compression methods described in the previous section. In Table 5 we show sizes of n-gram inverted file indexes for unsorted lexicons for n from 2 to 4, using V_G encoding. This method was used here because in an unsorted lexicon the run lengths for any n-gram are effectively random, and the V_G code is well suited to the resulting geometric distribution. In all of the tables in this section, index sizes include: the space required for an $N \cdot n$ byte array of N n-grams; one $\lceil \log_2 I \rceil$-bit pointer from each n-gram to its index entry, where I is the size of the set of index entries; the compressed index entries themselves; and, for each word in the lexicon, a $\lceil \log_2 C \rceil$-bit pointer (where the lexicon is C characters long) so that a 'word number' can be converted to a string for checking with *regex*. We also show, for $n = 3$, the time to create the index using an technique similar to the in-memory method described by Moffat for databases on secondary storage [14]. As n was increased, the creation times grew more slowly than did the size of the index, and times were dominated by the need to process the lexicon, which is independent of n.

By way of comparison, the sizes of the uncompressed indexes for *TREC* for n of 2, 3, and 4 are 277.9%, 259.5%, and 266.8% respectively, assuming $\lceil \log_2 W \rceil$ bits to represent each of the W ordinal word numbers.

	n	*Bible*	*Macq*	*TREC*
Total	2	113.9	102.7	119.9
index	3	174.3	139.8	158.0
size (%)	4	283.8	201.8	209.9
Creation time (sec)	3	7.4	36.9	608.9

Table 5: Sizes of n-gram indexes for unsorted lexicons

Or, from another perspective, the use of compression has reduced the space for each word number in the inverted index from 20 bits to about 10 bits.

This table shows a steady growth in index size with increase in n. A significant part of this growth, after $n = 2$, is due to the need to store the n-grams and pointers: for $n = 4$ the n-grams and corresponding search structure account for about 25% of the space requirement. This is partly because the number of n-grams is growing, and partly because we have not compressed them. Using prefix-omission, for example, the space the 4-grams require can be about halved, saving about 10% of the vocabulary size, but adding to the searching complexity. Even more effective would be the use of a minimal perfect hash function on the n-grams [5, 7]. This would allow the $N \cdot n$ space required by the n-grams to be reduced to about $4n$ bits, at little or no cost in lookup time. We did not explore this option, but it would be worth considering for a production im-

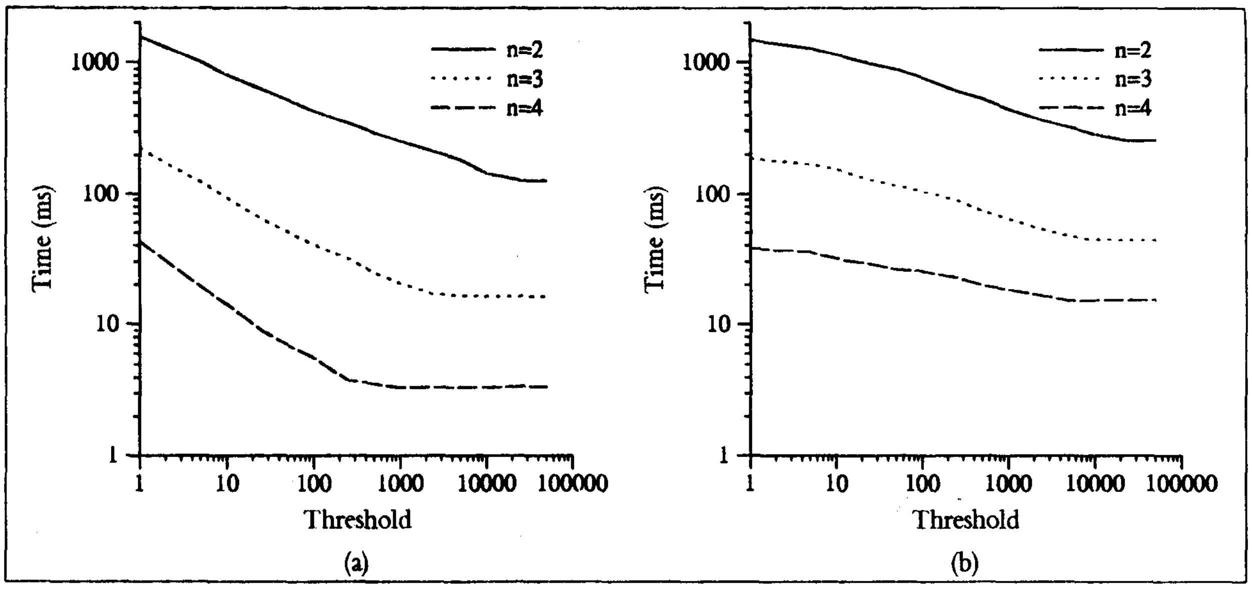

Figure 2: Effect of threshold on retrieval time for *TREC*: (a) full; (b) part

plementation on a static lexicon.

We show average evaluation times over the query set of 250 patterns in Table 6. All of these times are for unsorted lexicons and Elias's δ encoding, which is the slowest of the codings we have described and therefore gives an upper bound to retrieval time. These timings incorporate a minor optimisation: index entries for n-grams are processed in increasing order of length, so that throughout the sequence of mergings the number of candidate answers is kept as small as possible.

	n	*Bible*		*Macq*		*TREC*	
		part	*full*	*part*	*full*	*part*	*full*
Eval.	2	13.9	9.2	124.4	111.9	1,489.9	1,579.3
times	3	2.3	1.2	20.5	16.9	188.8	226.1
(ms)	4	0.6	0.3	4.9	3.8	38.1	43.1

Table 6: Speed of n-gram indexes for unsorted lexicons

These figures show a clear trade-off between size and speed, and this first implementation shows interim performance between the extremes of *regex* on the one hand and permuted lexicons on the other. Nevertheless, when compared to the permuted lexicon scheme, neither the space nor the time performance is particularly impressive. In the following sections we show how both can be substantially improved.

5.2 Improving performance: Thresholding

Noting that the cost of checking for false matches is low in this application, we next investigated methods by which it might be possible to trade relatively expensive index processing against relatively cheap false match checking.

A simple modification of this kind is to use a fixed *threshold*, and when the number of candidate answers falls below this threshold, no further index entries are merged. All of the remaining candidates are then accessed immediately and false matches eliminated with *regex*. This optimisation is effective because short entries are merged first, so that subsequent entries tend to be long and may not substantially reduce the number of candidates. That is, when longer entries are being processed costly decoding is being used to little effect, and so the returns are doubly diminishing. The impact of the use of thresholds on retrieval times is shown in Figure 2, where, for each combination of n and query set, average retrieval time is plotted as a function of the threshold value.

As can be seen, the improvement in retrieval time is dramatic, with matches found up to ten times faster and surprisingly large thresholds proving effective. The effect is particularly marked for fully-specified queries, which have many n-grams, the more common of which may provide no filtering at all. The different collections had, for our query set, different 'best' thresholds, which were, very roughly, around 1%–2% of lexicon size, and good performance was generally seen for a wide range of thresholds around the 'best' mark. Indeed, one simple heuristic would be to take as the set of candidates the words listed in the shortest inverted file entry, and use *regex* on all words that contained this single most discriminating n-gram.

5.3 Improving performance: Blocking

Another modification that trades index processing against false match checking is blocking of lexicon entries. In our description of the inverted file indexing scheme, each lexicon entry was indexed and allocated a unique number. However, the index will be smaller if adjacent entries are grouped into blocks. For blocked lexicons, the index into the lexicon would contain the ordinal numbers of the blocks containing each n-gram. The size decrease is both because some n-grams will occur more than once in a block but only require one reference in the index, and because the run lengths will be smaller and can be represented in fewer bits.

To answer a query, each of the blocks containing all of the n-grams is fetched, then searched to see if it contains any words that match the query. Figure 3 gives an example of a blocked n-gram index.

A simple blocking scheme is to divide the ordinal number of each lexicon entry by a fixed *blocking factor* B to give the ordinal number of the containing block so that each block contains B entries; a similar scheme was suggested by Owolabi & McGregor [19]. This method allows the lexicon entries to be stored individually, and a simple deblocking step is required to find individual words. This simple blocking scheme also allows the index pointers into the compressed inverted entries to be stored more economically, since only one pointer per block will be required, a total of $N\lceil\log_2 I\rceil/B$ bits rather than the previous $N\lceil\log_2 I\rceil$ bits. In Figure 4 we show the effect of this method of blocking on *TREC*. In this figure, each curve shows the space-time tradeoff gained by varying B from 1 (at the right-hand side, because indexes with block size of 1 are large) to 1024 (at the left-hand side).

As can be seen, there is an almost continuous trade-off between space and time performance—as block sizes get larger, less space is required for the index, but access is slower. Figure 4 uses a threshold of 1000, and in this case the trade-off between space and time is monotone. With a threshold of 1 the curves were actually 'bathtubs'—as B grew from 1, both space and time decreased until, at $B = 16$ or $B = 32$, the time started to grow again.

Table 7 summarises the gains that have been achieved by these two optimisations. Indexes of similar size to those of Table 6 have better time performance. For example, for *TREC* the new 4-gram index is both smaller and faster than the old 3-gram index. These results are a substantial improvement on those of Owolabi & McGregor, who, for an index of roughly 90% (in our framework) and a lexicon of 20,000 words, require 0.5 to 1 second per query on a Sun 3/60 [19]; on the same hardware, our implementation requires about 40 ms per query for a lexicon and index of this size.

	n	*Bible*		*Macq*		*TREC*	
Total	2	35.8		24.9		36.8	
index	3	103.7		68.1		80.7	
size (%)	4	220.8		136.8		138.8	
		part	*full*	*part*	*full*	*part*	*full*
Eval.	2	15.3	7.1	102.6	48.9	934.4	646.5
times	3	4.2	1.6	23.0	5.7	153.2	40.7
(ms)	4	2.0	0.4	19.8	2.5	129.9	9.8

Table 7: Index performance for unsorted lexicons, V_G compression, block size 16, threshold 1000

Other blocking schemes are possible. Blocks could be fixed length, containing a variable number of words. The suitability of such a scheme would depend on the kinds of access required to the lexicon. Alternatively, words with a large proportion of n-grams in common could be clustered into blocks, regardless of block size.

5.4 Sorted, static lexicons

In the previous section we considered optimisations that could be applied to arbitrary lexicons. If the lexicon is sorted—which, because of the complexity of update, is only feasible if it is static—further optimisations apply.

Indexes for sorted lexicons should compress well because short run lengths can be represented in only a few bits. In particular, a run length of 1 might be represented in as little as 1 bit, and in sorted lexicons it is the norm for adjacent words to share several n-grams. Moreover, the effect of blocking will be more pronounced in this case, because of the greater frequency of repetition of n-grams within blocks and the corresponding decrease in the number of false matches.

Another space optimisation is that prefix-omission can be used within blocks, and the characters of the remaining strings can be compressed. The block pointers can also be compressed, as they can be represented as a series of run lengths. We have not quantified the savings that these techniques would yield, but they will

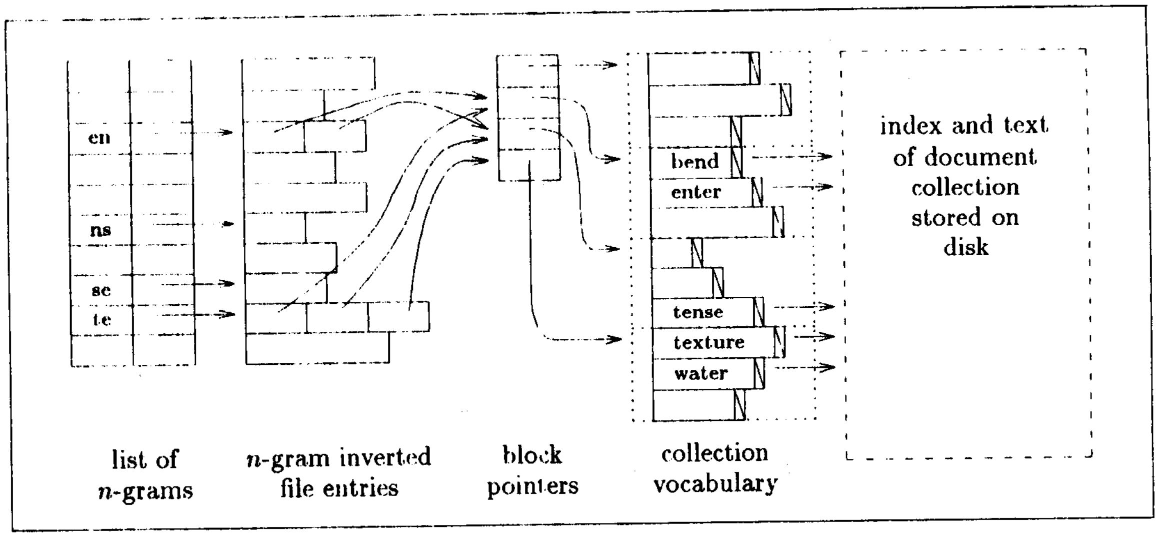

Figure 3: In-memory n-gram index with blocking, $B = 3$

be in addition to any saving yielded by other optimisations.

Binary search on the lexicon can yield a significant time saving. The sequence of characters up to the first wild card, for example the sequence **fro** in the pattern **fro*n**, can be used to identify a range in which all matches must lie, and there is no need to examine the index entries for the n-grams in this prefix sequence. In the limit, for a fully specified query, there is no need to examine the index at all. This time optimisation also yields a space optimisation: there is no need to index the first n-gram (for example, the 3-gram **|fr** in **frozen**) in each string. Although this implies that around 10% of index entries can be discarded, the space saving is in fact marginal, as run lengths of 1 can be represented in 1 bit and as a consequence these index entries are very small.

In Table 8 we show performance of n-gram indexes for sorted lexicons, for a block size of 16 and a threshold of 1000; these figures can therefore be contrasted directly with Table 7. We give sizes for the V_T encoding, which was the superior scheme in this case, and timings for δ. As can be seen, the indexes for sorted lexicons are at least 15% smaller than indexes for unsorted lexicons, and query evaluation is much faster. Note that the 2-gram indexes require even less space than the simple list of pointers (section three of Table 3), a rather remarkable testament to the efficacy of the compression methods employed.

	n	*Bible*		*Macq*		*TREC*	
Total index size (%)	2	23.6		16.6		20.4	
	3	76.2		41.8		46.9	
	4	175.7		98.7		96.0	
		part	*full*	*part*	*full*	*part*	*full*
Eval. times (ms)	2	4.9	0.1	41.8	0.1	229.2	0.4
	3	0.8	0.1	7.4	0.1	61.2	0.4
	4	0.3	0.1	2.8	0.1	49.6	0.4

Table 8: Index performance for sorted lexicons, V_T compression, block size 16, threshold 1000

It is worth noting the best times achieved for *part* queries and each collection: 0.31 ms on *Bible*, with n of 4, threshold of 10, a block size of 1, and an index of 244.2%; 2.1 ms on *Macq*, with n of 4, threshold of 100, a block size of 1, and an index of 171.0%; and 12.2 ms on *TREC*, with n of 4, threshold of 1000, a block size of 1, and an index of 170.1%. These figures compare well to the permuted lexicon scheme.

6 Conclusions

We have considered the compressed inverted file approach that has been previously applied only to secondary storage databases, and shown it to be viable in the main memory environment. Our indexing unit

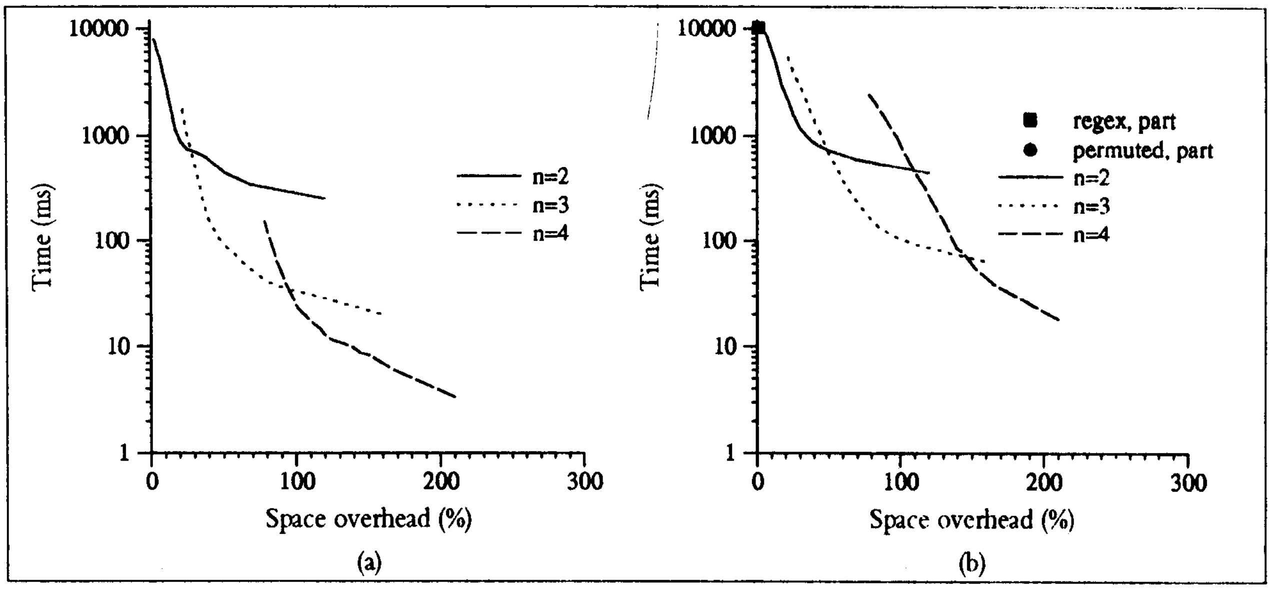

Figure 4: Effect of blocking on space and time for *TREC* and threshold of 1000: (a) full; (b) part.

has been n-grams. While there is no startling novelty in n-gram indexing, our figures for uncompressed, unoptimised n-gram indexing show that performance is unacceptably poor. It is the use of compression, thresholding, and blocking that results in the superior performance we have obtained in our experiments, allowing fast access to even large vocabularies such as that of the multi-gigabyte *TREC* collection. The experiments that we have undertaken are quite unambiguous in their results—that, suitably implemented, n-gram indexing runs orders of magnitude faster than brute force search on typical lexicons, and requires less main memory than traditional fast search structures such as permuted lexicons and tries. It thus offers a third alternative to the system designer charged with choosing a mechanism to support partial specification of query terms.

The techniques we have described are applicable to both sorted and unsorted lexicons, an advantage compared with the extended lexicon approach of Bratley & Choueka [4]. Our techniques also require substantially less memory. Moreover, our indexes can be built very quickly using an in-memory technique [14].

We can obtain a smaller, faster representation for sorted lexicons than for unsorted, and using a prefix-omission technique the words of the lexicon itself can be stored in less space if it is sorted. Thus, for a static database, we would prefer a sorted lexicon. Sorted lexicons also permit rapid binary search for matching strings when any of the initial letters of the string are provided. For more typical applications in which the database (and hence the lexicon) is dynamic, an unsorted lexicon is preferable so that new words can be easily inserted.

The single most important contributor to the success of our techniques has been the low cost of false match checking. This has allowed us to tolerate false match rates that would be unthinkable for a secondary storage database, and so at best the index need only be a crude filter eliminating most of the non-matching records. This flexibility permits the use of n-grams for small n, and allows us to employ blocking to reduce both the size of the index and the average query times. It would be surprising if similar trade-offs were not possible in other main memory databases. Moreover, if main memory costs continue to fall faster than secondary storage costs, we may see the advent of an age in which main memory databases are the rule rather than the exception. In this case the techniques we have described will prove invaluable.

Acknowledgements

We would like to thank Abe Bookstein, Andrew Hume, Alan Kent, and Tomi Klein for their advice and helpful discussion. We would also like to thank the anonymous referees for making many useful suggestions. This work was supported by the Australian Research Council.

References

[1] A.V. Aho and M.J. Corasick. Fast pattern matching: An aid to bibliographic search. *Communications of the ACM*, 18(6):333-340, 1975.

[2] A. Bookstein and S.T. Klein. Generative models for bitmap sets with compression applications. In *Proc. 14'th ACM-SIGIR Conference on Information Retrieval*, pages 63-71, Chicago, 1991.

[3] A. Bookstein, S.T. Klein, and D.A. Ziff. A systematic approach to compressing a full-text retrieval system. *Information Processing & Management*, 28(5), 1992.

[4] P. Bratley and Y. Choueka. Processing truncated terms in document retrieval systems. *Information Processing & Management*, 18(5):257-266, 1982.

[5] G.V. Cormack, R.N.S. Horspool, and M. Kaiserwerth. Practical perfect hashing. *Computer Journal*, 28(1):54-55, February 1985.

[6] P. Elias. Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, IT-21:194-203, March 1975.

[7] E.A. Fox, L.S. Heath, Q. Chen, and A.M. Daoud. Practical minimal perfect hash functions for large databases. *Communications of the ACM*, 35(1):105-121, January 1992.

[8] A.S. Fraenkel and S.T. Klein. Novel compression of sparse bit-strings—Preliminary report. In A. Apostolico and Z. Galil, editors, *Combinatorial Algorithms on Words, Volume 12*, NATO ASI Series F, pages 169-183, Berlin, 1985. Springer-Verlag.

[9] R.G. Gallager and D.C. Van Voorhis. Optimal source codes for geometrically distributed alphabets. *IEEE Transactions on Information Theory*, IT-21(2):228-230, March 1975.

[10] S.W. Golomb. Run-length encodings. *IEEE Transactions on Information Theory*, IT-12(3):399-401, July 1966.

[11] G. Gonnet and R. Baeza-Yates. *Handbook of data structures and algorithms*. Addison-Wesley, Reading, Massachusetts, second edition, 1991.

[12] ISO. *Commands for interactive text searching*, 1988. Draft International Standard ISO/DIS 8777.

[13] E.M. McCreight. A space-economical suffix tree construction algorithm. *Journal of the ACM*, 23(2):262-272, April 1976.

[14] A. Moffat. Economical inversion of large text files. *Computing Systems*, 5(2):125-139, 1992.

[15] A. Moffat and J. Zobel. Coding for compression in full-text retrieval systems. In *Proc. IEEE Data Compression Conference*, pages 72-81, Snowbird, Utah, March 1992. IEEE Computer Society Press, Los Alamitos, California.

[16] A. Moffat and J. Zobel. Parameterised compression for sparse bitmaps. In *Proc. ACM-SIGIR International Conference on Research and Development in Information Retrieval*, pages 274-285, Copenhagen, Denmark, June 1992. ACM Press.

[17] D.R. Morrison. PATRICIA—Practical algorithm to retrieve information coded in alphanumeric. *Journal of the ACM*, 15(4):514-534, 1968.

[18] National Institute of Standards and Technology. *Proc. Text Retrieval Conference (TREC)*, Washington, November 1992. Special Publication 500-207.

[19] O. Owolabi and D.R. McGregor. Fast approximate string matching. *Software—Practice and Experience*, 18:387-393, 1988.

[20] C.E. Shannon. A mathematical theory of communications. *The Bell Systems Technical Journal*, 27:379-423, 1948.

[21] J. Teuhola. A compression method for clustered bit-vectors. *Information Processing Letters*, 7(6):308-311, October 1978.

[22] E. Ukkonen. Approximate string matching with q-grams and maximal matches. *Theoretical Computer Science*, 92:191-211, 1992.

[23] I.H. Witten, T.C. Bell, and C.G. Nevill. Indexing and compressing full-text databases for CD-ROM. *Journal of Information Science*, 17:265-271, 1992.

[24] J. Zobel, A. Moffat, and R. Sacks-Davis. An efficient indexing technique for full-text database systems. In *Proc. International Conference on Very Large Databases*, pages 352-362, Vancouver, Canada, August 1992.