Comparing Inverted Files and Signature Files for Searching a Large Lexicon

BEN CARTERETTE[1], FAZLI CAN[2]
Computer Science and Systems Analysis Department
Miami University, Oxford, OH 45056; December 4, 2003
To appear in *Information Processing and Management*

Abstract
Signature files and inverted files are well-known index structures. In this paper we undertake a direct comparison of the two for searching for partially-specified queries in a large lexicon stored in main memory. Using *n*-grams to index lexicon terms, a bit-sliced signature file can be compressed to a smaller size than an inverted file if each *n*-gram sets only one bit in the term signature. With a signature width less than half the number of unique *n*-grams in the lexicon, the signature file method is about as fast as the inverted file method, and significantly smaller. Greater flexibility in memory usage and faster index generation time make signature files appropriate for searching large lexicons or other collections in an environment where memory is at a premium.
Keywords: *Compression, Dictionaries, Indexing Methods, Personal Digital Assistants (PDAs), Performance Evaluation.*

1. Introduction

Searching a large lexicon is a fundamental activity in information retrieval: the first step in resolving a query to a document collection index is finding query terms in a lexicon (Witten et al., 1999; Baeza-Yates & Ribeiro-Neto, 1999). Lexicons, being relatively small, can be stored in main memory and searched very fast, but it is worth considering the gains that indexing the lexicon separately might give. A lexicon index could allow for *partially-specified query terms* (e.g. terms with a wildcard character representing multiple unspecified characters) by pattern matching.

Partially-specified terms have uses in cross-language retrieval (Guthrie et al., 1996), spell checking (Kukich, 1992), approximate matching (Zobel & Dart, 1994), crossword puzzle generation (Harris et al., 1993; Harris et al., 1992), and library catalog retrieval (Crane, 1996), to name a few. Another application is query expansion: a preprocessing step could use a lexicon search to translate a pattern into a set of terms for a disjunctive search. Glimpse (Manber & Wu, 1993) uses approximate matching for file system search. Partial and approximate matching have

[1] Now at the Center for Intelligent Information Retrieval, University of Massachusetts, Amherst, MA 01003. e-mail: carteret@cs.umass.edu.
[2] Corresponding author, Computer Science and Systems Analysis Department, Miami University, Oxford, OH 45056, e-mail: canf@muohio.edu, voice: +1 (513) 529-5950, fax: +1 (513) 529-1524.

turned out to be especially useful on hand-held computers (Personal Digital Assistants, PDAs) such as Palm Pilots (Kaljuvee et al., 2001; Buyukkokten et al., 2002). A method better than brute force searching and matching is needed if partial term resolution is to be a frequent activity, especially for the aforementioned PDA. An index to the lexicon is needed for maximum efficiency.

Much research has been done into two indexing methods—inverted files and signature files. Most of the research has been focused on searching a full-text database. Inverted files have been used to search a large lexicon for partially-specified terms (Zobel et al., 1993). Enhancements to increase the speed of signature file processing include a multiorganizational scheme (Kent et al., 1990), bit-slicing (Roberts, 1979; Witten et al., 1999), vertical framing (Koçberber & Can, 1997), horizontal partitioning (Zezula et al., 1991), tree structures (Tousidou et al., 2000) and multi-level superimposed coding (Lee et al., 1995), among others (Aktug, Can, 1993; Faloutsos, 1992). Inverted files have been compared to signature files for searching a full-text database (Zobel et al., 1998), with inverted files claiming victory. A key point of that comparison was that the performance of signature files suffers because records in a full-text database vary greatly in length. Since the length of records in a lexicon is more uniform, we hypothesize that a signature file index will compare favorably to an inverted file index for searching a lexicon. Our primary intent in this paper is to compare the two for this purpose.

Our motivation in this study is to index and search a large lexicon for partially-specified queries using signature files, which has not been done before, and to show, by mathematical argument and experimental comparison, that a signature file approach can be as good as or better than an inverted file approach for this application. Specifically, we will show that signature files are about as fast as inverted files while using less memory for the index and allowing more flexibility in index size, which is important in an environment in which memory is at a premium. We also address some of the criticisms of signature files by Zobel et al. (1998) by showing that run length compression is useful for reducing the size of a signature file and by comparing the two in terms of flexibility and extensibility. We believe our work will open the door to the use of signature files for indexes in low-memory environments.

The paper is organized in the following way. Section 2 presents previous work on partial matching, including the inverted file method used by Zobel et al. (1993). Section 3 summarizes signature file research and the signature file method used for comparisons. In Section 4 we will discuss our test data, test queries, and the two programs used in the comparison, and in Section 5 we undertake a direct comparison of the two methods, first by reasoning, then by mathematical

modeling, and finally by experimentation. In Section 6 we discuss other bases of comparison and consider the application of our results to the hand-held environment, and Section 7 is the conclusion.

2. Previous Work

Before considering inverted files and signature files for indexing a lexicon, we will look at other approaches to searching for partially-specified queries.

Bratley and Choueka (1982) propose a permuted dictionary in which each term is "cyclically shifted," using wrapping to generate the string starting at each character in the word. A termination character is appended to the word to mark its boundary. The word `file`, for example, would contribute to the dictionary the terms `file|`, `ile|f`, `le|fi`, `e|fil`, and `|file`. The dictionary is sorted alphabetically to allow fast binary search. Partially-specified queries can be found by a simple query conversion (`X*` becomes `|X`; `*X` becomes `X|`; `X*Y` becomes `Y|X`, where `X` and `Y` are substrings) and dictionary search. The obvious drawback of this approach is the space requirement. Bratley and Choueka propose a prefix-omission technique, but the overhead is still large.

Owolobi and McGregor (1988) use *n-grams,* sequences of *n* consecutive characters in words, to index terms. The parameter *n* can be any integer greater than one; for example, the 2-grams that comprise `information` are `in`, `nf`, `fo`, `or`, `rm`, `ma`, `at`, `ti`, `io`, and `on`. Processing partially-specified query terms is accomplished by extracting the *n*-grams from the query and finding the terms that contain those *n*-grams. In Owolabi and McGregor's case this is done with a bitmap matrix. A query on `*inf*` would find words that contain the 2-grams `in` and `nf` and return the conjunction of those sets. The word `information` would be returned as a match, as would `reinforces`.

This turns out to be a very useful method for indexing a lexicon (Robertson & Willett, 1998; Zobel et al., 1993; Witten et al. 1999). It requires considerably less space than the permuted index and is much faster than brute force matching. One problem with *n*-grams is that the possibility of *false matches* arises: in the example above, `confine` would also be among the returned words, as it too contains the 2-grams `in` and `nf`. False matches can be reduced, but not eliminated, by the inclusion of a start-of-word character `^` and an end-of-word character `$`, granting each term two additional *n*-grams. Despite the false matches, the work of Zobel et al. (1993) convinces us that *n*-grams are the best method for indexing a lexicon, and it is the method that we will use in this work.

Zobel et al (1993) implement a compressed inverted file in which each *n*-gram has a posting list of terms in which it appears. Inverted files are a well-known index structure usually consisting of an *access structure* for a set of terms drawn from a collection of documents and *posting lists* of pointers to the documents that terms appear in. When *n*-grams are used to index terms, the analogy to document indexing is that the terms become "documents" and *n*-grams become "terms". Here we will summarize the inverted file method used for lexicon search.

When a query to an inverted file is made, the query's *n*-grams are extracted and located in the access structure. The posting list for each *n*-gram is retrieved, and the conjunction of the lists constitutes the set of potential matches. Each potential match must be checked against the query to ensure it is true match. The UNIX library *regex* can be used to resolve matches by regular expression matching (Other algorithms exist and may be faster; for instance, the *nrgrep* algorithm of Navarro (2001)) See Figure 1 for an example of query resolution with an inverted file.

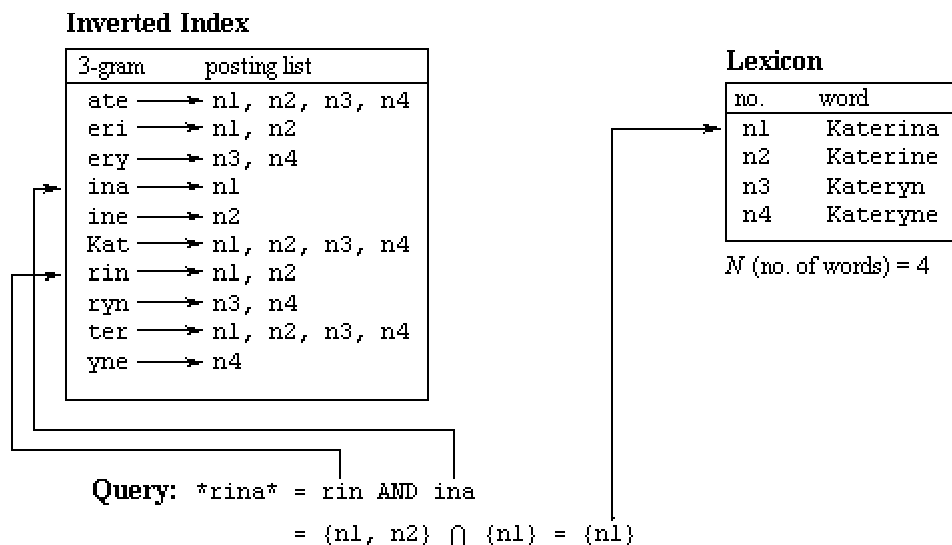

Figure 1. Query resolution with an inverted file.

Posting lists can grow very long in a large lexicon. Run length encoding can be used to compress the lists. A *run length* is the distance between two pointers in a list. For example, the posting list for an *n*-gram that occurs in words (1, 5, 10, 13, 17, 22, 50, 57, 58, 60) could be represented as run lengths (1, 4, 5, 3, 4, 5, 28, 7, 1, 2). The run lengths expose patterns that can be used for compression. The standard method for compressing inverted files is the δ run-length encoding of Elias (1975).

If space is at a premium, the inverted lists can be stored as pointers to the start of a block of records instead of as pointers to individual records. In this case, a *blocking factor B* is chosen,

and terms are grouped into blocks of *B* terms each. The posting list for an *n*-gram contains pointers to the block number in which a term that contains that *n*-gram appears; as long as *n*-grams appear more than once in a block, memory is saved in the posting list lengths. When the lists are retrieved and conjoined, all of the terms in all of the matching blocks must be tested against the query term to resolve false drops. Therefore there is an expected tradeoff between index size reduced by blocking and the increased number of false drops.

Skips, or *synchronization points*, can be added into an inverted list (Moffat & Zobel, 1996). A synchronization point is a point in the list at which decoding can begin; they exist so that the irrelevant parts of an inverted list can be skipped over and just the relevant portions decompressed, ideally saving some time in list processing. Specifically, if there are to be *p* synchronization points, then each inverted list has an index of *p* pointers to positions in the compressed list where decompression can begin.

For example, the list (1, 5, 10, 13, 17, 22, 50, 57, 58, 60) could be given four synchronization points. The skip list would be (1, 13, 50, 60). If another list has already been processed and it is known that the query has no answers between terms 13 and 50, then the original list only needs to be decompressed up to term 13 and after term 50—7 run lengths to be decompressed instead of 10. An additional list of address pointers ($<a_1>$, $<a_2>$, $<a_3>$, $<a_4>$) is required to specify the bit position of the compressed list at which to start decompression. To decompress the original posting list after term 50, the address pointer list is decompressed to find $<a_3>$, the bit position in the compressed posting list corresponding to the run length between terms 50 and 57. Decompression commences at that point. The necessity of two additional lists (skip list and address pointer list) increases index size, but both lists can be compressed with run length compression to save space.

Since false match resolution (in our case by using *regex*) is so fast, only a few of the lists associated with the *n*-grams need to be decompressed and merged. A limit can be placed on the number of matches, and lists can be processed until that limit, the *threshold*, is reached, at which point false match resolution commences. The optimal threshold can be found using the ratio of the time it takes to process a posting list to the time it takes to resolve a false match (see section 5.2.2). If the inverted lists are processed in order of increasing size, the most discriminating *n*-gram lists are processed first, and the threshold can be reached very quickly. Sorting the lists by size is a negligible processing cost that is more than offset by the time saved in false match resolution and list processing.

3. Signature Files

A *signature* is a bit mapped abstraction of a record. There are two main methods of generating signatures: word signatures and superimposed coding (Faloutsos & Christodoulakis, 1987). In the word signature approach, identifiers (words, or *n*-grams in our case) of a record are hashed to bit patterns—word signatures—which are later concatenated to form the record signature. The superimposed coding method hashes each unique identifier to S bit positions in a bit string with a fixed width F and superimposes (via bitwise OR) the resulting signatures to generate the record signature (some of the frequently used symbols of the paper are listed with their definitions in Table 1). An example superimposed signature for the word `Katerina` based on 3-grams with F=16 and S=2 is shown in Figure 2. We use the superimposed coding method for this study.

```
     Kat   0000 0100 0000 0001
     ate   0001 1000 0000 0000
     ter   0000 0000 1000 0100
     eri   0100 0001 0000 0000
     rin   0000 0100 0001 0000
     ina   0010 0010 0000 0000
--------------------------------
Katerina   0111 1111 1001 0101
```

Figure 2. Superimposed signature generation for `Katerina` based on 3-grams.

In a naïve signature approach, hashing is random and uniform, and any given *n*-gram always hashes to the same S bit positions. There is a chance that two different *n*-grams will hash to the same bit position; this is referred to as a *collision*. Since we choose F much less than the total number of unique *n*-grams, collisions are possible.

A *sequential signature file*, or SSF, is made up of a series of signatures, one for each record to be indexed. A query is processed by generating its signature (via the same process used to generate record signatures) and comparing it to each signature in the SSF by bitwise AND. The matching records are retrieved using an address table; they then must each be compared to the query because some of them can be false matches due to collisions.

Table 1. Table of symbols

symbol	definition
B	Number of terms in a block
b	Average number of unique *n*-grams in a block
D	Average number of unique *n*-grams in a term
d	Number of *n*-grams in a given term
F	Signature width in bits
FD_i	Number of possible matches after processing *i* bit slices
h	Threshold
i	Number of slices to be processed in query resolution
N	Number of unique terms in the lexicon
S	Number of bits set per *n*-gram in a term
T	Total query time
t_{slice}	Time required to process a bit slice
$t_{resolve}$	Time required to resolve a match

To improve processing time, a signature file can be processed by slices. This is referred to as a *bit-sliced signature file*, or BSSF. Conceptually, the signature file is a matrix with N rows (the number of records) and F columns. A column a in the matrix is similar to a posting list in an inverted file: it represents a list of records in which the identifier that hashed to bit position a might appear. To resolve a query, the columns corresponding to on-bits in the query signature are conjoined to give the set of possible matches. Matches are retrieved using an address table and compared against the query. See Figure 3 for a diagram of query resolution with a BSSF.

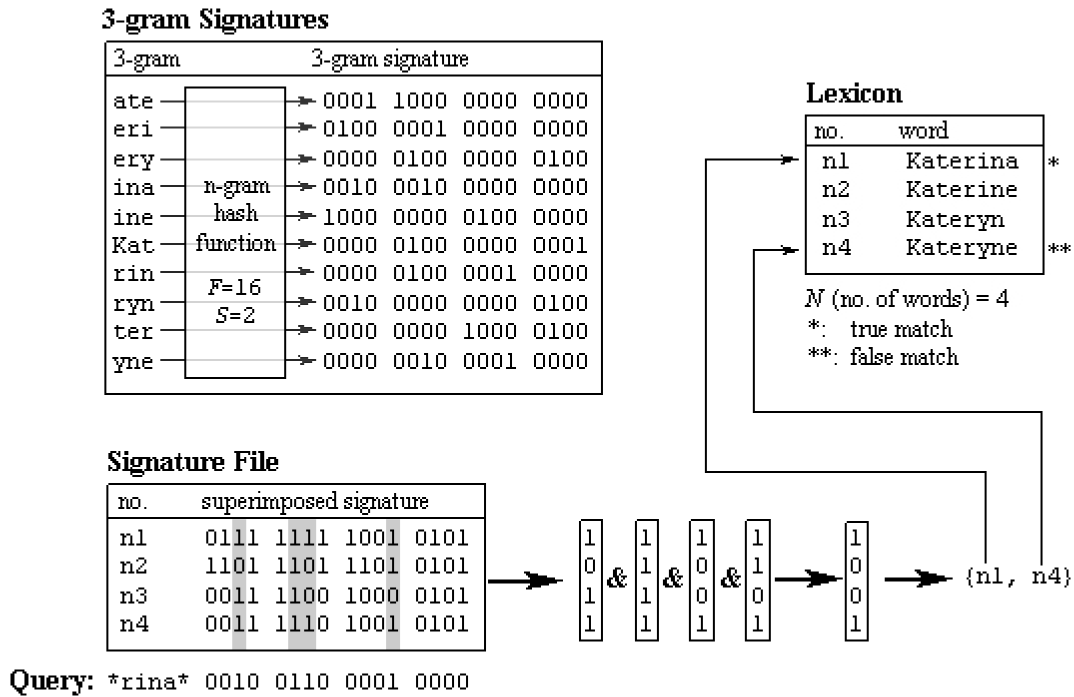

Figure 3. Query resolution with a bit-sliced signature file.

In this study BSSF is the “Signature File” structure that we have chosen to measure the signature file performance; for brevity we will use SF to indicate BSSF in the rest of the article.

A signature file for a large lexicon can be relatively large. A lexicon with 232,435 terms and a signature width of 1,000 would yield a 28MB signature file. A signature file can be compressed by the same run length encodings described earlier, and it will compress well given a long and sparse signature.

Compression is only viable for a signature file if its density (measured by the proportion of on-bits to total bits) is very low. If density is too high, the rate of compression will be low, and decompression will be slow. Signature files have fewer run lengths than inverted files due to collisions, and since there are many fewer slices in a signature file than lists in an inverted file, the size of the compressed signature file is expected to be less than that of the compressed inverted file.

If space is at a premium, the same blocking method described above for an inverted file can also be applied to a signature file. A signature would represent a block of B terms instead of a single term. Collisions would be somewhat more likely, as more n-grams are combined in a single signature; as a result, compression would be slightly better. Overall processing time when blocking is used will increase, as there will be many more false matches.

Because of the similarity of a bit slice to an inverted list, bit slices can be skipped just like posting lists. As with an inverted file, for each bit slice there is a skip list and a list of address pointers. After processing the first bit slice, skip lists are used to save processing time by only decompressing the pieces of bit slices that need to be examined.

Above we discussed thresholding as a method of partially evaluating a query to an inverted file index. Bit-sliced signature files can also be partially processed. Slices are processed (decompressed and bit wise ANDed) until the *stopping condition* is satisfied. The stopping condition dictates that bit slice processing should stop after i bit slices if the time it would take to process the $(i+1)^{st}$ bit slice plus the resulting set of matches FD_{i+1} is greater than the time it would take to process the current set of matches FD_i (Koçberber & Can, 1996). With a sparse signature file and fast false match resolution, as little as one slice can narrow down the lexicon to a small set of possible matches.

We have decided that it is best to keep it simple. Other signature file methods attempt to reduce false matches, but the tradeoff is more computationally-expensive processing of bit slices. Since matches in a lexicon can be resolved so quickly using *regex*, a query can return many false drops but still be processed quickly. We have therefore focused on keeping the time required to process a bit slice low.

4. Experimental Design and Data

4.1. Lexicons

We culled our lexicons from three sets of documents: the United States Code, the federal laws available from `http://uscode.house.gov/download.htm`; all United States Supreme Court opinions (decisions, concurrences, and dissents) from 1893 to 2002, downloaded from findlaw.com; and articles from the *Financial Times* from the TREC 4 CD.

The words were extracted from the collections of documents as follows: First, all markup tags were removed. The Supreme Court cases were marked up with HTML and the TREC database with SGML; everything between and including a less-than and a greater-than was removed. All non-alphanumeric characters were removed, so for instance *don't* became *dont* and *baseball-only* became *baseballonly*. This was done on the assumption that someone might want to search on a term like *baseball-only*, and that removing the hyphen and contracting the terms would narrow the search more than a conjunctive query on *baseball* and *only*. Numbers were not removed on the assumption that someone might want to search for a section number, date, monetary value, etc. Of course since non-alphanumeric were removed, a date like *1/11* (January

11 or November 1) becomes *111* and indistinguishable from section *1.11*, which also becomes *111*. There was no modification of case, so *Frank* is distinguished from *frank*.

4.2. Lexicon Statistics

The U.S. Code lexicon (henceforth referred to as *uscode*) has 232,435 unique terms. It uses 1.7MB of disk space. The average length of a term is about 6.69 characters. The Supreme Court lexicon (*scotus*) has 372,760 unique terms and uses 3.1MB of disk space. The average term length is about 7.81 characters. The lexicon from *Financial Times* articles from TREC (*ft*) has 803,400 unique terms and uses 7.1MB of disk space. The average term length is about 8.25 characters. Table 2 shows these numbers along with length distribution statistics and the number of unique *n*-grams for each sorted lexicon for *n*=2, 3, 4, and 5.

Table 2. Lexicon Statistics

	uscode	*scotus*	*ft*
size on disk (MB)	1.7	3.1	7.1
terms	232,435	372,760	803,400
avg term length	6.69	7.81	8.25
max term length	44	180	63
std. dev.	2.87	3.32	2.86
2-grams	3,212	3,502	3,706
3-grams	33,942	42,949	56,170
4-grams	129,909	184,077	257,658
5-grams	273,267	447,113	675,519

Some 99.5% of the terms in *uscode* are 20 characters long or less, and the other two lexicons are similar. Since the length of terms does not vary much compared to a collection of documents like the *Wall Street Journal* articles used by Zobel et al. (1998), this is an ideal application for a naïve signature file. The signature width can be chosen such that the number of on-bits is a low percentage of total bits, giving each term a unique signature. Long records indexed along with short records adversely affect the performance of signature files, since those long records are much more likely to be returned as possible matches and take longer to resolve (Zobel et al., 1998). In a lexicon, the records are of a more uniform length; very long terms are rare enough so as not to noticeably affect performance.

Non-naïve signature file approaches have been used to index documents of non-uniform length with a small amount of space overhead. See (Koçberber & Can, 1997; Koçberber et al., 1999).

4.3. Binary Search

Binary search is a fast algorithm for searching for strings in a sorted lexicon. Because of the possibility of queries that begin with a wildcard, binary search cannot be used alone, but it may be combined with an index to speed up processing. Queries that start with ^ can quickly be reduced to a range of potential matches. For example, the three characters `^fo` in the query `^fo*th*` narrow the number of possible results in *scotus* to 1,208, from `fo` to `foyers`. With enough characters specified, the index structure is bypassed entirely. We elected to use only queries that would access the index structure since a binary search is equally fast regardless of the indexing method.

Furthermore, if binary search is to be used, *n*-grams starting with ^ do not need to be indexed. This saves some space in an inverted index and reduces the number of collisions in a signature file. It will also save some space in a compressed signature file.

4.4. Queries

Query terms were randomly selected from the *ispell* dictionary used by UNIX systems for spell checking. It consists entirely of English words and thus is a good representation of normal queries to a lexicon search. Full queries can be processed quickly with a binary search; the speed with which partial queries are processed is the true measure of the system. Partial queries were generated by replacing a random number of sequences of random numbers of characters with the wildcard character `*`. Each query had at least one *n*-gram so that no brute force matching (which would take equally long for inverted files or signature files) would be done. About half the queries used the end character `$`. We generated two query sets: query set `TWO` is a collection of 100 queries with an average of two 3-grams per query; no query in this set has more than three 3-grams. Query set `SIX` is a collection of 100 queries with an average of six 3-grams per query; all queries in this set have five, six, or seven 3-grams. Each query ran 100 times to get a good average for processing time. Table 3 shows examples from both query sets.

Table 3. Example query terms

TWO		SIX	
*r*sume*	*Ba*d*in$	*swi*ingly$	*car*lessness*
ation	*all*	*gainer*	*smuggle*
*ker$	*colt*	*guished$	*colon*ally$
*orc*d*	*gen*al$	*section*	*indire*t*ng$
*t*ing$	*co*ue$	*Luxem*ourg$	*per*ndicul*rs*

4.5. Description of Programs

For our comparison, we used the inverted file program described by Zobel et al (1993) and available online at `ftp://ftp.cs.rmit.edu.au/pub/rmit/fnetik/src/vrank`. We modified the program to do signature processing. The inverted file method implemented by the program can be seen as a signature file method with minimal perfect hashing and a signature width equal to the number of unique *n*-grams. Likewise, the signature file method we use can be seen as an inverted file method with imperfect hashing and fewer lists. They are programmatically very similar. We have made both programs (including a bug fix in the inverted file code—see Note at the end of this work) available on our website at `http://cs.umass.edu/~carteret/bssf.html`.

The inverted file program used for query processing stores *n*-grams in a hash table along with their inverted lists. Our signature file program uses an array to store the F bit slices. Hashing of *n*-grams is done on the fly, so no *n*-gram hash table is needed. It may in fact be possible to increase the speed of signature files by finding a faster hash function. See, for instance, (Cohen, 1997). When signature width was fixed at compile time, the signature file program reported slightly faster times than when it was dynamically allocated at runtime. We used the latter, but in an environment in which F is known beforehand and reindexing will be very rare, it might make sense to fix the array size.

Tests were done on a dual-processor 1266MHz Pentium III with 512MB of RAM and a 512K cache. The machine is running Linux 2.4.17. Tests were done with both sorted and unsorted lexicons, with binary search enabled on the sorted lexicons (making the indexes slightly smaller). The threshold point was the same for both programs. Results given throughout this paper are for sorted lexicons indexed using 3-grams, unless otherwise noted. The use of 3-grams follows Zobel et al. (1993), Owolabi and McGregor (1992), and others who have found 3-grams to give a good ratio of speed to indexing overhead (Adams & Meltzer, 1993; Angell et al., 1983).

Although our results have implications for indexing in environments such as PDAs with limited memory availability and slow processing time, implementing full inverted file and signature file systems on a PDA is beyond the scope of this work. Complications involved in a full implementation include the necessity of using data structures more suited to the Palm environment (recursion is nearly impossible because of the small stack size (Noble & Weir, 2001)) and the necessity of writing code to minimize heap size use first and memory use second (Rhodes & McKeehan, 1999). In Section 6, we attempt to indicate how our results can be mapped to the Palm environment by considering the efficiency of pieces of the algorithms.

5. Comparison

5.1. Direct Comparison by Reasoning

The argument against signature files being faster than inverted files is that bit slices are longer than inverted lists and thus take longer to process; that more slices will have to be processed than inverted lists processed; and that there will be more false matches (Zobel et al., 1998). This is all true, but we contend that for searching a large lexicon (or any collection of documents with close to uniform length), the number of slices that needs to be processed is only slightly larger; that the number of matches to be resolved is only slightly more; and that slices are on average only slightly longer, so that the extra time required because of extra processing is miniscule. Furthermore, because there are fewer bit slices in a signature file than inverted lists in an inverted file, the relevant slices can be accessed faster, decreasing the overall time and making the difference between the two negligible.

In terms of space, the argument in favor of inverted files is that they can be compressed while signature files cannot (Zobel et al., 1998), which is true for the signature file structure described in the study. However, it cannot be generalized for all signature file structures; in fact, the effect of compression on signature files has been studied (Faloutsos & Christodoulakis, 1987; Faloutsos, 1992) and shown to be beneficial. As long as the signatures are sparse, run length compression works well. However, there are fewer bit slices in a signature file than posting lists in an inverted list, and the signature file does not require an access structure for *n*-grams, so the signature file is smaller. Also, because signature width is variable, a signature file can be reindexed to fit into a very small amount of memory if needed. This is discussed in more detail in section 6.

Another argument against signature files is that they are more expensive to generate (Zobel et al., 1998). This is a one-time cost, so perhaps not that important, but because bit slices can be generated *sui generis* (the signature matrix does not need to be generated and then transposed), it takes less time to generate a signature file index than an inverted file index. There are fewer lists to be added to as processing continues, and the lists are accessed faster by hashing on the fly instead of searching an access structure for an *n*-gram.

5.2. Direct Comparison by Mathematical Modeling

Our intent in this section is to modify the mathematical model presented elsewhere (Koçberber, 1996; Koçberber et al., 1999) for our lexicon search application, and to show that for this application, inverted files are a special case of signature files.

5.2.1. A Mathematical Model for Signature Files

The parameters in a basic signature method are shown in Table 1. They are *F*, the width of the signature; *S*, the number of on-bits set by each *n*-gram; and *D*, the average number of *n*-grams in a term. The signature method described here also uses parameters *B*, the blocking factor, *b,* the number of unique *n*-grams in a block, and *h*, the threshold. The variables t_{slice} and $t_{resolve}$ are the time required to process a slice and the time required to resolve a match (i.e., to determine whether it is a true match or a false match). Equations for signature file evaluation have been derived (Roberts, 1979; Koçberber & Can, 1996); we use slightly modified equations here that take term blocking into account. The density of the signature file, its *op* value, is estimated probabilistically as $1-\left(1-S/F\right)^b$. Note that this estimate of *op* value assumes that there will be no collisions in the *S* bits set per *n*-gram, but that there may be collisions in the $S \cdot b$ bits set by the *b* *n*-grams in a block. Note also that when *B*=1 it follows that *b=D.* The expected number of potential matches remaining after processing *i* slices is estimated as $FD_i = N \cdot op^i = N \cdot \left(1-\left(1-S/F\right)^b\right)^i$.

The total expected time it will take to process a query of *d* *n*-grams using a signature file index is:

$$T_i = t_{slice} \cdot i + t_{resolve} \cdot FD_i \qquad (1)$$

where *i* is the number of slices to be processed, at maximum $S \cdot d$.

Note that *op* is a function of *S*, *F*, and *b* (which itself is a function of *D* and *B*). The parameters *S, F,* and *B* are set by the user when the lexicon is indexed; the parameter *D* is directly related to *n*-gram length. It is trivial to show that as *F* increases, *op* decreases (and vice versa), and that as *S* or *B* increase, *op* increases (and vice versa). The time t_{slice} is a function of *op* and *N.* As *op* increases, t_{slice} increases, and vice versa. Likewise for *N*. Sparser, shorter bit slices process faster. Overall, as *F* increases, query time *T* decreases (Figure 4).

The time $t_{resolve}$ is a function of the complexity of the query. Queries with more wildcard characters take slightly longer to match; if full regex querying is enabled, $t_{resolve}$ could vary greatly from query to query. On average, $t_{resolve}$ is very fast—timed at about .002 milliseconds for partial queries.

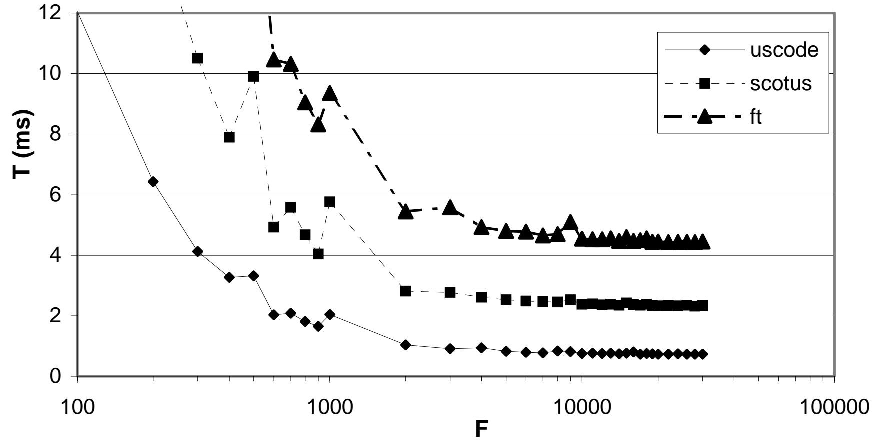

Figure 4. Query time *T* vs. *F (S= 1)*.

Although false match resolution is fast, a low false drop rate is desirable for fast overall processing. A false drop rate of 1 in 100,000 is stated mathematically as:

$$op^i = \left(1 - \left(1 - \frac{S}{F}\right)^b\right)^i = \frac{1}{100000}$$

where i is the number of slices to be processed. Solving for i gives

$$i = \frac{\ln\left(\frac{1}{100000}\right)}{\ln(op)} = \frac{\ln\left(\frac{1}{100000}\right)}{\ln\left(1 - \left(1 - \frac{S}{F}\right)^b\right)} \quad (2)$$

In some cases i slices cannot be processed. The number of slices that can be processed depends on the weight of the query. The weight of a query with d n-grams is given as

$$W_Q(d) = F \cdot \left(1 - \left(1 - S/F\right)^d\right) \quad (3)$$

At least one slice must be processed; at most $W_Q(d)$ slices can be processed. In other words, i is bounded:

$$1 \le i \le W_Q(d) \le S \cdot d$$

The above equations are useful for finding parameters that give good results. Note that *op* remains approximately the same when *S* and *F* are increased by the same factor. The time to process a slice also remains approximately constant, as seen in Figure 5. If query weight is low, processing $W_Q(d)$ slices may not eliminate enough false drops to achieve a good query time. In that case, increasing *S* and *F* by the same factor may decrease query time significantly. Specifically, *S* and *F* should be increased by the same factor if the following inequality is true.

$$t_{resolve} \cdot \left(N \cdot op^{W_Q(d)} - N \cdot op^{W_Q(d)+1} \right) \geq t_{slice} \tag{4}$$

In other words, if the time it would take to process the false matches that could be eliminated by adding another on-bit to the query signature while keeping *op* constant is greater than the time it takes to process a slice, *S* and *F* should be increased in such a way that *op* remains constant. This is only good to a point, as *F* cannot be larger than the number of unique *n*-grams. Also, if *op* is sufficiently small, or if $W_Q(d)$ is sufficiently large, the difference in the number of false drops achieved by increasing *S* and *F* is small and not worth the attendant increase in index structure size. The *op* is sufficiently small when *F* is large and *S* is small. An *op* of 1 in 1000 is small enough that *S* can be fixed at one.

For an *op* of less than one in 1000, equation 2 gives

$$i = \frac{\ln\left(\frac{1}{100000}\right)}{\ln(op)} < \frac{\ln\left(\frac{1}{100000}\right)}{\ln\left(\frac{1}{1000}\right)} = \frac{5}{3}$$

Fewer than 1.67 slices will need to be processed per query to achieve a false match rate of one in one hundred thousand. An *op* of .00035 (achieved for *uscode* when *F*=17,000) would only require 1.45 slices to be processed per query.

We introduced the concept of skipping earlier, and consider it now. Before any skip lists can be processed, one full slice must be decompressed and processed. Because we predict that fewer than two slices will be processed on average, it is unlikely that skipping will have a beneficial effect. Even in cases where more than two lists need to be processed, the additional time required to decompressed the skip list and address pointers will likely offset the time gained by skipping. When the signature file is dense, however, skipping can reduce processing time. The signature file may be dense if *F* is small, *S* is large, or *B* is large. More slices need to be processed in a dense signature file; each additional slice that is processed reduces the number of possible matches and therefore increases the chance that sections of the next slice can be skipped.

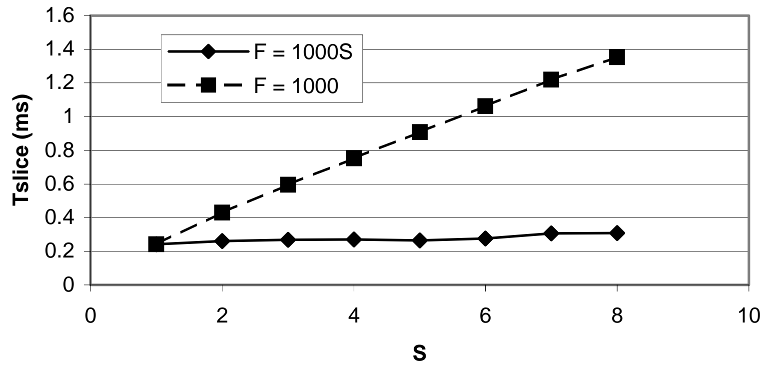

Figure 5. t_{slice} **vs.** S **for** ***uscode*** **with** F **constant and** $F = 1000S$.

The optimal threshold h is based on the formal stopping condition, which says to stop processing when the estimated time it would take to process another bit slice and resolve the resulting set of matches is greater than or equal to the time it would take to resolve the current set of matches. Mathematically,

$$t_{slice} + t_{resolve} \cdot N \cdot op^{i+1} \geq t_{resolve} \cdot N \cdot op^{i}$$

By rearranging terms and noting that $N \cdot op^{i} \cdot (1 - op) \approx N \cdot op^{i}$, processing should stop when:

$$\frac{t_{slice}}{t_{resolve}} \geq N \cdot op^{i} \qquad (5)$$

i.e., when the ratio of t_{slice} to $t_{resolve}$ is greater than or equal to the number of terms in the current result set. The threshold value can be based on an estimate or a timing of the two values rather than timing them during program execution.

5.2.2. A Mathematical Model for Inverted Files

The inverted file index can be considered a signature file index with minimal perfect hashing and F equal to the number of unique n-grams. In that case, the op value of that index is approximately equal to $\frac{b}{F}$. Using equation 2 gives the number of slices that will need to be processed to achieve a false drop rate of 1/100000, and in fact, the number we come up with is remarkably close to the number we see in experiments.

$$op^{i} = \left(\frac{b}{F}\right)^{i} = \frac{1}{100000}$$

$$i = \frac{\ln\left(\frac{1}{100000}\right)}{\ln(op)} = \frac{\ln\left(\frac{1}{100000}\right)}{\ln\left(\frac{b}{F}\right)}$$

In the case of *uscode*, where F=33942 and b=5.69 without blocking, the *op* value is less than 1/5000, which means the number of lists that will need to be processed is about 1.35. Again, it is unlikely that the skip lists will be helpful, since so few slices need to be processed.

Equation 5 applies for determining the optimal threshold. Therefore the threshold for inverted files will be $t_{slice}/t_{resolve}$. Although the inverted file is less dense, experiments show t_{slice} for an inverted list to be close to t_{slice} for a sparse bit slice (specifically, within 1% difference).

Now we have shown that the number of slices processed by a signature file is only slightly larger than the number of lists processed by an inverted file; and that the number of matches will be similar since the optimal thresholds are identical. Experimentation will verify that query time is similar and that the signature file index size is smaller.

5.3. Experimental Comparison

In this section we present selected results from the experiments described in section 4. Section 5.3.1 gives results using 3-grams, which give the best ratio of query time to space overhead for inverted files (Zobel et al., 1993). In section 5.3.2 we explore using longer or shorter *n*-grams.

5.3.1. Experiments using 3-grams

Table 4 and Table 5 show the best observed query times along with the average number of slices processed per query and the average number of matches resolved (true and false) per query for each lexicon for both inverted files and signature files. The signature width was 17,000, well under the number of unique 3-grams. Both tables show that signature files do only slightly more processing per query than inverted files. The inverted file is faster than the signature file, but only by microseconds.

Table 4 shows results for query set `TWO` on sorted lexicons. A sorted lexicon is searched faster than an unsorted lexicon, since in the latter, the 3-grams starting with ^ are not indexed (see Section 4.3). Although the unsorted lexicon is slower for both SF and IF, the percent difference is about the same. Short queries typically return many potential matches; resolving them is a large percentage of the total processing time.

Table 4. Average query performance with query set `TWO` on sorted lexicons

	uscode		*scotus*		*ft*	
	SF	IF	SF	IF	SF	IF
slices or lists	1.45	1.42	1.65	1.64	1.78	1.77
matches	411.38	407.25	1136.82	1123.51	1862.48	1839.44
time (ms)	1.282	1.255	4.005	3.920	5.707	5.567
% time difference	2.11%		2.12%		2.45%	

Table 5 shows results for query set `SIX` on unsorted lexicons. With longer queries there is a greater chance of a sparse bit slice or short list being used, so there are fewer potential matches. Resolving those accounts for less of the overall processing time; a greater percentage of processing is done in decompressing and merging lists or bit slices. The inverted file is faster, because of its shorter lists, but the difference is only in tens of microseconds.

Table 5. Average query performance with query set `SIX` on unsorted lexicons

	uscode		*scotus*		*ft*	
	SF	IF	SF	IF	SF	IF
slices or lists	1.38	1.28	1.89	1.80	2.12	2.04
matches	30.03	29.19	34.17	34.01	33.70	32.49
time (ms)	.090	.082	.344	.330	.470	.440
% time difference	8.89%		4.07%		6.38%	

Table 6 shows the sizes on disk and in memory of the indexes that gave the best observed times for both methods for each lexicon. The signature file index is significantly smaller in every case. The signature file index size is the sum of the sizes of the compressed slices. Once again, signature width was 17,000. The access structure size for the signature file includes a pointer to each bit vector, plus the space required for extra information about each bit vector (its length, current position, and so on), plus the size of the pointers to the terms in the lexicon. The inverted file index size is the sum of the sizes of the compressed inverted lists. The access structure size for the inverted file includes the space required for the unique n-grams, the size of the hash table nodes that the n-grams are stored in, and the size of the pointers to the terms in the lexicon. (Calculation of access structure size is shown in more detail in Section 6.1). The percentage by which the inverted file size exceeds that of the signature file is given in the final row. The signature file is significantly smaller while being nearly as fast.

Table 6. Size of index structures as a percentage of lexicon size for sorted lexicons

	uscode		*scotus*		*ft*	
vocab size (MB)	1.70		3.13		7.09	
	SF	IF	SF	IF	SF	IF
index	71%	75%	76%	78%	66%	68%
access	82%	126%	62%	96%	50%	73%
total size	154%	201%	138%	174%	117%	141%
(MB)	(2.61)	(3.41)	(4.31)	(5.44)	(8.29)	(9.99)
% size difference	31%		26%		21%	

Table 7 shows the index generation time for the indexes that gave the best observed query times. The signature file index is generated faster in every case. Of course, query time is the most important measure; we will not attempt to argue otherwise. But note that inverted files do not have a very large advantage there; the difference is less than 3% for short queries and less than 10% for long queries.

Table 7. Time to generate index structures for sorted lexicons

	uscode		*scotus*		*ft*	
	SF	IF	SF	IF	SF	IF
time	7.78s	11.65s	15.29s	23.48s	32.68s	48.35s
% difference	50%		54%		48%	

5.3.2. Experiments with other gram lengths

Using longer *n*-grams gives interesting results. It is expected that increasing *n* will speed up query processing, as there are more *n*-grams and each one appears in fewer terms, and that space overhead will increase, as more bit slices or inverted lists are needed to accommodate the increase in *n*-grams. The results for inverted files conformed to our expectation, but the results for signature files were surprising: query times equally good or better with higher values of *n* in the same amount of space overhead. The numbers in Table 8 were obtained using a query set made up of 10,000 five-letter strings, so the queries had four 2-grams, or three 3-grams, or two 4-grams, or one 5-gram, depending on the length of the *n*-gram used for indexing.

Table 8. Space overhead as a percentage of lexicon size and query time for sorted *uscode*

	n=2	n=3	n=4	n=5
signature width	9000	2000	2000	3000
space overhead	120%	121%	124%	124%
query time (ms)	.758	.272	.219	.230

The reason good query times are still obtained without increasing space overhead is that although there are many more 4- and 5-grams than 3-grams, each 4- or 5-gram occurs in far fewer terms than 3-grams. The average number of terms a 4- or 5-gram appears in is less than the average number of terms a 3-gram appears in, and the variance is less. Thus bit slices are more uniformly dense, and the *op* value used to optimize query time more closely reflects the actual density of the slices being processed. In the context of our mathematical model, the *op* is lower for higher values of *n* because of the parameter *b* in its calculation.

This also explains why the observed numbers for 3-grams are greater than the predicted numbers. The predicted numbers are based on slices of uniform density; the observed numbers were obtained using queries taken from English words, which means the density of each slice processed was greater than the average density of the signature file.

Table 9 compares signature files and inverted files for different gram lengths. Signature width was chosen low for *n*=2, because *F* cannot be larger than the number of unique *n*-grams. See Table 2 for the number of unique *n*-grams in each lexicon. The memory usage for inverted files is very large because of the additional *n*-grams.

Table 9. Space overhead and query time for sorted *uscode*

	n=2		*n*=4		*n*=5	
	SF	IF	SF	IF	SF	IF
signature width	3000	-	30000	-	40000	-
total space overhead	109%	112%	198%	465%	206%	843%
query time (ms)	.848	.758	.046	.032	.034	.017

6. Other Points of Comparison

In this section we will compare signature files and inverted files using some of the criteria set out by Zobel et al. (1996). Specifically, we will compare them in terms of memory usage, scalability, index generation time, ease of update, and extensibility.

6.1. Memory Usage

Both methods store the entire lexicon and the index structure in memory. The inverted file method additionally must store the lookup table. For the *uscode* lexicon, with 33,942 3-grams, the lookup table adds 101,826 bytes. The 8 bytes required by each 3-gram for the hash table plus the 28 bytes needed for each bit vector bring the total amount of space required by the lookup table to 1,293KB. Adding in a 4-byte pointer for each term (*N*=232,435; see Table 2) in the lexicon (they are stored in an array of strings) brings the total to 2,234KB. The signature file method needs the 4-byte pointer for each term, the 28 bytes for each bit vector, and 4-byte

pointers for storing the bit vectors in an array. Since there are many fewer bit vectors in the signature file, the amount of additional memory required before considering the term pointers is 531KB when the number of bit slices is 17,000—half the number of lists in the inverted file. The total amount of overhead is 1,439KB.

Table 10. Storage overhead for *uscode* with *B*=1 (sizes in bytes)

	IF		SF
a. total length of posting lists	1,331,273	a. total length of bit slices	1,284,511
b. # of lists	33,942	b. # of slices	17,000
c. *n*-gram length	3		
d. hash node size	8	c. array pointer size	4
e. vector overhead	28	d. vector overhead	28
total = a+b*(c+d+e)	2,655,011	total = a+b*(c+d)	1,828,511

The biggest cost in either approach is the lexicon and its pointers. The lexicon may be compressed using methods described by Faloutsos (1985), Bell et al. (1989), or Navarro et al. (2000). Compressing the lexicon would presumably incur a tradeoff in processing time as lexicon terms would need to be decompressed, or query terms represented in some compatible way, before matching. Another way to reduce overhead is by storing lexicon terms continuously and using a $\lceil \log_2 C \rceil$-bit pointer (where C is the total number of characters in the lexicon) to access them.

Not counting lexicon overhead, the inverted file for *uscode* requires 2,593KB, while the signature file requires 1,786KB, a 31% difference. Table 10 shows how index overhead is calculated.

Table 11. Size in megabytes of index structures under various conditions

block size	lexicon	IF	SF F=17000	SF F=100
1	*uscode*	2.53	1.74	0.72
	scotus	4.03	2.89	1.35
	ft	6.90	5.21	2.76
	total	13.46	9.84	4.83
20	*uscode*	1.98	1.19	0.17
	scotus	2.93	1.78	0.28
	ft	4.64	2.95	0.60
	total	9.55	5.92	1.05

While memory on modern desktops is cheap and plentiful, memory usage may be extremely important in a PDA environment. Typical PDAs have 8-32MB of memory. Table 11 shows the amount of space our three index structures use under various conditions. Storing the three inverted file structures for our three lexicons would use 13.46MB, 37% more than the three

signature files at 9.84MB (at *F*=17,000). The ability to shrink the index structure could be a huge bonus. The total size of the inverted files could be reduced to 9.55MB by using a blocking factor of 20; the signature file size is reduced to 5.92MB using the same blocking factor. The signature file size could be reduced to 4.83MB using a signature width of 100 and no blocking factor (query time is much slower, of course, but still measured in milliseconds). Finally, using a signature width of 100 and a blocking factor of 20, the combined size of the three signature files would be only 1.05MB. The capability to shrink an index structure to any size could be invaluable to the international businessman or foreign diplomat.

On a PDA, the lexicon and lexicon index would be stored in permanent memory, in the database structure. One database record can hold up to 64KB; the average inverted list is 39 bytes and the average bit slice is 76 bytes. On average, then, all inverted lists or bit slices could be stored in about 20 records to minimize storage overhead (each database record requires some additional overhead containing information about the record (Palm, 2003)). The inverted files also must store the *n*-grams and some way of associating a list with a *n*-gram, which would require at least one more record. Palm OS 4.0 has 182KB of dynamic heap space available for processing (Palm, 2003), which is more than enough to decompress and process several lists. The lexicon can span records, but would require an additional structure within each record to point to term heads.

6.2. Scalability

We have demonstrated the scalability of signature files by showing results from lexicons with 232,435 terms to lexicons with 803,400 terms. Time differences appear to favor signature files as the lexicon gets bigger, due to the increase in the number of unique 3-grams. The advantage signature files hold in memory usage shrinks as lexicon size increases due to the greater percentage of total space consumed by the access structure. If the access structure can be shrunk, perhaps using methods mentioned in section 6.1, the advantage of the signature file index remains high.

Generally as lexicon size increases query time increases. In the query time equation 1, t_{slice} and $FD(i)$ are functions of N. Index size is expected to rise with lexicon size, as is generation time. The same things will happen to an inverted file index.

It is worth considering whether the comparison scales downward; i.e. whether the results will hold in the low-memory hand-held environment. For query resolution time, the points of comparison are the amount of additional time needed by the inverted file to find an *n*-gram in its hash table; the additional time required by the signature file to decompress bit slices; and the

additional time required by the signature file to resolve false matches. Again, it is beyond the scope of this work to quantify these points, but if we assume proportionality, the percent difference in processing time will remain the same.

6.3. Index Generation Time

While index generation time will usually be a one-time cost, adding a lexicon to a PDA may require that the other lexicons stored on the PDA be reindexed for memory usage.

With each *n*-gram setting one bit in the signature (i.e., S=1), the number of run lengths that will be compressed while creating the signature file index is slightly less than the number of run lengths that will be compressed while creating the inverted file index, due to collisions. Furthermore, *n*-grams can be hashed on the fly during signature file generation, saving the time required to look them up in a structure.

Parameter choice affects index generation time as follows: as F increases, generation time increases. As S increases, generation time increases. As B increases, generation time decreases. Indexes for larger lexicons are generated slower, as are indexes for lexicons with more *n*-grams per term.

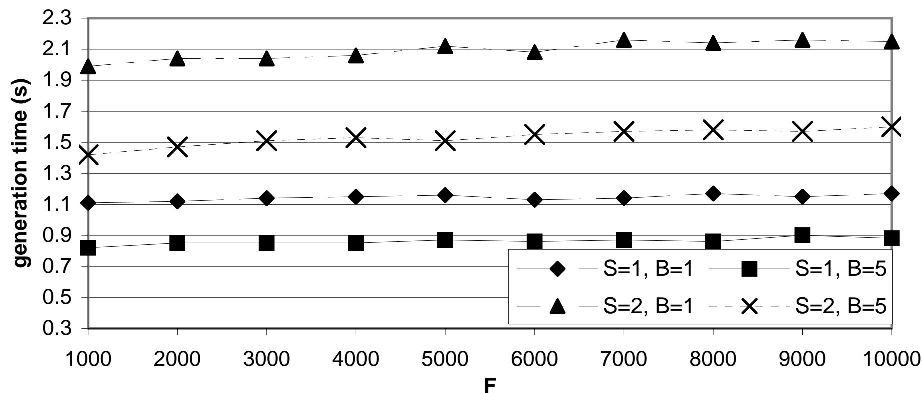

Figure 6. Index generation time vs. *F* for *uscode*.

It should be noted that increasing S has a huge effect on generation time. Doubling S almost doubles generation time. Therefore for this analysis to hold, signatures need to be long enough that S can be fixed at one. On the other hand, increasing F has a very small effect on generation time. See Figure 6.

Palm OS 4.0 has only 256KB of dynamic heap space, 72KB of which are reserved for system use (Palm, 2003). This makes indexing a large lexicon in memory infeasible. In a production application, this would probably be moved to the desktop machine, where databases

would be indexed and then uploaded. The signature file application retains the ability to index in memory if needed, provided the number and size of bit slices is low enough to fit in memory. Indexing can also be done within the Palm's permanent database structure, although it would presumably be much slower.

6.4. Index Update

To add a term to an unsorted lexicon, the run lengths for the *n*-grams in the term are compressed and appended to the relevant posting list or bit slice. A small amount of overhead is needed to lengthen the bit vector. In either inverted files or signature files, the update will be fast. It may be slightly faster in signature files because there is no need to look up the *n*-grams in a table; if the added term has an *n*-gram that is not in the inverted file lookup table, it needs to be added and the lookup table rewritten to disk.

Previous literature has asserted that insertions to a sorted bit-sliced signature file is practically impossible, due to the need to decompress the entire signature file, add the new signature in the correct spot, regenerate the slices, and recompress each slice (Zobel et al., 1998; Croft & Savino, 1988). While it is true that all slices must be decompressed and recompressed, inserting to an inverted file entails the same overhead. It is not the case that slices need to be regenerated, as run length encoded bit slices are essentially the same as inverted lists: an inverted list is a list of terms an *n*-gram appears in, while a bit slice is a list of terms an *n*-gram appears in combined with some terms it does not appear in. In either case insertion is accomplished by splitting a currently existing run length in two. In fact, other forms of signature compression, such as VBC (Faloutsos & Christodoulakis, 1987), allow for partial decompression and recompression.

We also consider that because signature files are shorter (due to collisions) and thus compress faster, because there are fewer bit slices than inverted lists, and because *n*-grams do not have to be located in a lookup table, signature files will have an advantage in insertion time.

6.5. Extensibility

Signature files are extensible as long as inaccuracy is acceptable. The inaccuracy can be minimized with appropriate selection of parameters at index time.

Approximate matching, or ranking, can be useful in a lexicon search for a spell-check application or a translation dictionary. Partial query matching has been shown to be useful for approximate matching (Zobel & Dart, 1994). The *n-gram distance* algorithm seems to be one of the best available, providing another justification for our use of *n*-grams for indexing. The *n*-

gram distance algorithm works as follows: the bit slices or posting lists relevant to query *n*-grams are located and unioned (instead of intersected; we want all terms that have one *n*-gram in common with a query term). The *gram-dist* (Ukkonen, 1992) between the query term and a retrieved term is computed as the sum of the differences in the number of times each n-gram that occurs in both terms occurs in each term. If we assume that terms don't contain repeated *n*-grams (a reasonable assumption usually), the formula is $gram-dist(s,t)=|G_s|+|G_t|-2|G_s \cap G_t|$, where G_x is the set of *n*-grams in term *x*. For example, *gram-dist(file, filing)* is $2+4-2\cdot 1=4$ because they have one 3-gram in common *(fil)*. Terms are ranked from lowest *gram-dist* to highest. Zobel and Dart show that this algorithm is as good as or better than most others while being simple to calculate.

This is algorithm is easily implemented using inverted files, in which the *n*-gram count is equal to the number of on-bits in the inverted list. It can be implemented using a signature file with the following tradeoff: if greater accuracy is desired, processing time will be greater. If faster processing time is desired, accuracy will be lower. The *gram-dist* can be computed using bit slice lengths, but because bit slices are stochastic, the ranking will be inaccurate. For accuracy, the *n*-grams in each term can be found.

Research has been done into ranking methods with signature files; however, practical use of signatures for ranking has not been reported in the literature. Croft and Savino (1988) describe a method of ranking using bit-sliced signature files similar to an inverted file. Lee and Ren (1996) describe a method of partitioning signature files based on term weights. Their query evaluation method is sequential instead of bit-sliced, but it entails many fewer false matches. Luk and Chen (2001) extend VBC to support term frequencies. Their WVBC method is sequential as well, but a VBC-coded bit vector can be partially decompressed, saving processing time. Both of these methods reduce the problem of terms that are not similar to the query term.

A lookup table like that used in an inverted file can also be added to a signature file, if additional space overhead is acceptable. The nodes in the lookup table can hold additional information about *n*-grams.

It is true, however, that signature files are less extensible than inverted files. Signature files are essentially binary; they can only say whether or not a record has a specific property or not. Without additional storage overhead, they cannot compete with inverted files in this respect.

7. Conclusion

The lexicon search is a ubiquitous activity in any kind of information retrieval system and its index structure needs careful design and implementation. A lexicon is small enough to be stored in main memory, and using *n*-grams, partial query matching is efficient. We have shown by mathematical argument and experimental comparison that for searching a large lexicon, a signature file index is as good as an inverted file index. Although more slices need to be processed than (inverted index posting) lists, slices are longer than lists, and more false matches will need to be processed, the differences are small.

Efficient searching of a large lexicon is useful and important by itself, but the conclusion can be generalized. Our mathematical and experimental analyses above demonstrate that a signature file index can be as good as an inverted file index given a set of records and queries with the following properties:

1. The number of unique identifiers per record is small compared to the total number of unique identifiers, which is large, allowing sparse signatures.
2. Query weight is high relative to record length. This allows just one bit to be set per unique identifier. This property is conditional—if signatures are sufficiently sparse, query weight is irrelevant.
3. False drop resolution is fast relative to the time required to process lists or slices, forcing an inverted list to partially process queries to compete with signature files.

When these properties hold, the compressed signature file index will be smaller than the compressed inverted file index, it will be generated faster, and queries will be processed nearly as fast. Collections that might index well with signature files are library catalogs, multimedia files with many attributes, medical cross references, or lists of streets for a GPS system. Quantifying the conditions listed above would be useful in determining whether a signature file index is good for an application.

The most obvious and immediate implication of our results is the application to the low-memory environment of hand-helds such as Palms. While some of the basics of the Palm environment were discussed, complete production-level inverted file and signature file applications are needed to draw a meaningful conclusion. It is apparent that the ability of signature files to quickly reindex into smaller spaces would be very useful; since we predict and observe a small query time difference, see this as a worthy direction for future research.

Acknowledgment. We greatly appreciate the TREC 4 CD made available by NIST. We thank Justin Zobel for making his inverted file code freely available. We are very grateful to the referees, whose comments and suggestions greatly improved the focus of this work.

Note. A bug in the publicly-available inverted file code results in fewer matches reported when binary search is enabled. Partial queries that end with `$` are not processed correctly due to an error in pointer arithmetic. Specifically, line 175 in `mkvind.c` should read: `if (!dorange) len += 2; else len++`. The bug was corrected for our comparison.

References

Adams, E. and Meltzer, A. (1993). Trigrams as index elements in full text retrieval: Observations and experimental results. In *Proceedings of the 1993 ACM Conference on Computer Science*, pages 443–439.

Angell, R., Freund, G., and Willett, P. (1983). Automatic spelling correction using a trigram similarity measure. *Information Processing & Management*, 19(4):305—316.

Aktug, D. and Can, F. (1993). Signature files: an integrated access method for formatted and unformatted databases. Working Paper 93-006, Systems Analysis Dept, Miami University, Oxford OH.

Baeza-Yates, R., and Ribeiro-Neto, B. (1999). *Modern Information Retrieval.* ACM Press, New York.

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

Bell, T., Witten, I., and Cleary, J. G. (1989). Modeling for text compression. *ACM Computing Surveys*, 21(4):557–591.

Buyukkokten, O., Kaljuvee, O., Garcia-Molina, H., Paepcke, A., Winograd, T. (2002). Efficient web browsing on handheld devices using page and form summarization. *ACM Transactions on Information Systems,* 20(1):82—115.

Cohen, J. (1997). Recursive hashing functions for n-grams. *ACM Transactions on Information Systems*, 15(3):291–320.

Crane, G. (1996). Building a digital library: The Perseus project as a case study in the humanities. In *Proceedings of the First Annual Conference on Digital Libraries*, pages 3–10.

Croft, W. B. and Savino, P. (1988). Implementing ranking strategies using text signatures. *ACM Transactions on Office Information Systems*, 6(1):42–62.

Elias, P. (1975). Universal codeword sets and representations of the integers. *IEEE Transactions on Information Theory*, IT-21:194–203.

Faloutsos, C. (1985). Access methods for text. *ACM Computing Surveys*, 17(1):49–74.
Faloutsos, C. (1992). Signature Files. In W. B. FRakes and R. Baeza-Yates (Ed.) *Information Retrieval Data Structures and Algorithms*, (pp. 44–65). Prentice Hall, Englewood Cliffs, N.J.

Faloutsos, C. and Christodoulakis, S. (1987). Description and performance analysis of signature file methods for office filing. *ACM Transactions on Office Information Systems*, 5(3):237–257.

Guthrie, L., Pustejovsky, J., Wilks, Y., and Slator, B. M. (1996). The role of lexicons in natural language processing. *Communications of the ACM*, 39:63–72.

Harris, G., Forster, J., and Rankin, R. (1993). Basic blocks in unconstrained crossword puzzles. In *Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing*, pages 257–262.

Harris, G., Roach, D., Smith, P. D., and Berghel, H. (1992). Dynamic crossword slot table implementation. In *Proceedings of the 1992 ACM/SIGAPP Symposium on Applied Computing*, pages 95–98.

Kaljuvee, O., Buyukkokten, O., Garcia-Molina, H., Paepcke, A. (2001). Efficient web form entry on PDAs. In *Proceedings of the 10th World Wide Web Conference*, pages 663—672.

Kent, A. J., Sacks-Davis, R., and Ramamohanarao, K. (1990). A signature file scheme based on multiple organizations for indexing very large text databases. *Journal of the American Society for Information Science*, 41(7):508–534.

Koçberber, S. (1996). *Partial Query Evaluation for Vertically Partitioned Signature Files in Very Large Unformatted Databases*. PhD thesis, Bilkent University, Ankara, Turkey.

Koçberber, S. and Can, F. (1996). Partial evaluation of queries for bit-sliced signature files. *Information Processing Letters*, 60:305–311.

Koçberber, S. and Can, F. (1997). Vertical framing of superimposed signature files using partial evaluation of queries. *Information Processing and Management*, 33(3):353–376.

Koçberber, S., Can, F., and Patton, J. M. (1999). Optimization of signature file parameters for databases with varying record lengths. *The Computer Journal*, 42(1):11–23.

Kukich, K. (1992). Techniques for automatically correcting words in text. *ACM Computing Surveys*, 24(4):377–439.

Lee, D. L., Kim, Y. M., and Patel, G. (1995). Efficient signature file methods for text retrieval. *IEEE Transactions on Knowledge and Data Engineering*, 7(3):423–435.

Lee, D. L. and Ren, L. (1996). Document ranking on weight-partitioned signature files. *ACM Transactions on Information Systems*, 14(2):109–137.

Luk, R. W. P. and Chen, C. M. (2001). Document ranking for variable bit-block compression signatures. *Information Processing and Management*, 37(1):39–51.

Manber, U. and Wu, S. (1993). GLIMPSE: a tool to search through entire file systems. *Proceedings of the 1994 Winter USENIX Technical Conference*, pages 23—32.

Moffat, A. and Zobel, J. (1996). Self-indexing inverted files for fast text retrieval. *ACM Transactions on Information Systems*, 14(4):349–379.

Navarro, G. (2001). NR-grep: A fast and flexible pattern matching tool. *Software—Practice and Experience,* 31(13):1265—1312.

Navarro, G., de Moura, E. S., Neubert, M., Ziviani, N., and Baeza-Yates, R. (2000). Adding compression to block addressing inverted index. *Information Retrieval*, 3:49–77.

Noble, J., and Weir, C. (2001). *Small Memory Software: Patterns for Systems with Limited Memory*. Addison-Wesley, Harlow, England.

Owolabi, O. and McGregor, D.R. (1988). Fast approximate string matching. *Software—Practice and Experience*, 18(4):387—393.

Palm, Inc. (2003). *Palm OS Programmer's Companion*. Available online at `http://www.palmos.com/dev/support/docs/palmos/CompanionTOC.html`.

Rhodes, N. and McKeehan, J. (1999). *Palm Programming: the Developer's Guide*. O'Reilly & Associates.

Roberts, C. S. (1979). Partial-match retrieval via the method of superimposed codes. *Proceedings of the IEEE*, 67(12):1624–1642.

Robertson, A. and Willett, P. (1998). Applications of n-grams in textual information systems. *Journal of Documentation*, 54(1):48–69.

Tousidou, E., Nanopoulos, A., and Manolopoulos, Y. (2000). Improved methods for signature-tree construction. *The Computer Journal*, 43(4):301–314.

Ukkonen, E. (1992). Approximate string matching with *q*-grams and maximal matches. *Theoretical Computer Science*, 92:191—211.

Witten, I. H., Moffat, A., and Bell, T. C. (1999). *Managing Gigabytes: Compressing and Indexing Documents and Images, 2nd ed.* Morgan Kaufmann, San Francisco.

Zezula, P., Rabitti, F., and Tiberio, P. (1991). Dynamic partitioning of signature files. *ACM Transactions on Information Systems*, 9(4):336–367.

Zobel, J. and Dart, P. (1994). Finding approximate matches in large lexicons. *Software–Practice and Experience*, 25(3):331–345.

Zobel, J., Moffat, A., and Ramamohanarao, K. (1996). Guidelines for presentation and comparison of indexing techniques. *ACM SIGMOD Record*, 25(3):10–15.

Zobel, J., Moffat, A., and Ramamohanarao, K. (1998). Inverted files versus signature files for text indexing. *ACM Transactions on Database Systems*, 23(4):453–490.

Zobel, J., Moffat, A., and Sacks-Davis, R. (1993). Searching large lexicons for partially specified terms using compressed inverted files. In *Proceedings of the 19th VLDB Conference*, pages 290–301.