Computer Science and Systems Analysis

Computer Science and Systems Analysis

Technical Reports

Miami University *Year* 1999

Compressed Bit-sliced Signature Files An Index Structure for Large Lexicons

Fazli Can* Ben Carterette†

*Miami University, commons-admin@lib.muohio.edu

†Miami University, commons-admin@lib.muohio.edu

MIAMI UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE & SYSTEMS ANALYSIS

TECHNICAL REPORT: MU-SEAS-CSA-1999-001

Compressed Bit-sliced Signature Files
An Index Structure for Large Lexicons

Fazli Can and Ben Carterette

School of Engineering & Applied Science | Oxford, Ohio 45056 | 513-529-5928

Compressed Bit-sliced Signature Files
An Index Structure for Large Lexicons

by

Fazli Can and Ben Carterette
Systems Analysis Department
Miami University
Oxford, Ohio 45056

Working Paper #99-001 **04/99**

Compressed Bit-sliced Signature Files An Index Structure for Large Lexicons

Fazli Can *Ben Carterette*

Systems Analysis Department
Miami University
Oxford, OH 45056, USA
E-mail: canf(carterba)@muohio.edu
Tel: 1-(513) 529-5950 (4781)

ABSTRACT

We use the signature file method to search for partially specified terms in large lexicons. To optimize efficiency, we use the concepts of the partially evaluated bit-sliced signature file method and memory resident data structures. Our system employs signature partitioning, compression, and term blocking. We derive equations to obtain system design parameters, and measure indexing efficiency in terms of time and space. The resulting approach provides good response time and is storage-efficient. In the experiments we use four different lexicons, and show that the signature file approach outperforms the inverted file approach in certain efficiency aspects.

KEYWORDS: Lexicon search, n-grams, signature files.

INTRODUCTION

Information retrieval (IR) is a prevalent activity in today's information-oriented world. Systems ranging from a personal digital assistant (PDA) to a World-Wide Web search engine such as AltaVista need to be able to efficiently retrieve information from databases of small to hard-to-imagine sizes. In these systems queries are commonly executed in environments based on the Boolean or vector space models [SAL89]. For query:document matching most IR systems require a list of all terms (lexicon) used in the database. Depending on the user needs, the lexicon may contain the actual words as they appear in the documents or the word stems. After locating a query term in a lexicon, an IR system can easily locate the other information needed for the subsequent steps of query processing by using a pointer associated with individual lexicon words. Efficient lexicon indexing is crucial in overall system efficiency.

The most common file structures used in IR systems are inverted files and signature files [SALT89, WIT94]. In this study our database is a lexicon of actual words and we will develop a signature-based indexing scheme for efficient lexicon searches for partially specified terms.

Searching a lexicon for a partially specified term using the inverted file method involves a list of segments of terms (called *n-grams*) with a corresponding list of terms those *n*-grams appear in. When a user enters a query, the query term is split into its *n*-grams, and the lists of words that contain those *n*-grams are returned and merged. The common elements of these lists are the potential matches to the user's query term [ZOB93].

In this paper we use the superimposed signature file method [CHR84] (for the sake of brevity we will henceforth drop the adjective 'superimposed'). In the signature file method, each attribute of an object which describes the object is hashed into a bit string of size F by setting S bits to "1" (*on-bit*) where $S << F$. Object signatures are obtained by superimposing (bitwise *OR*ing) the signatures of object attributes. Within the context of this study, objects are terms and attributes are term n-grams. To minimize search time, the signature file is stored in main memory. In other words, it is not strictly a file anymore; however, we will follow traditional naming and use the term "file."

To optimize efficiency, we use a modified version of the Partially evaluated Bit-Sliced Signature File (PBSSF) method [KOC96a, KOC96b, KOC97]. Our approach employs partitioning, compression, and blocking within the framework of PBSSF. We derive equations to describe the system and obtain its design parameters. We measure system efficiency in terms of time and space. Pros and cons of our system are compared with the inverted file approach.

We start by describing the term n-gram and the principal concepts of signature files. We then present the basic signature file system we will be using Partially Evaluated Bit Sliced Files (PBSSF). We proceed to add partitioning,

compression, and blocking, in that order, to improve the overall performance of the system, interspersing these modifications with justifications and test results. Finally, we briefly introduce other approaches and compare our system to the inverted file system described by Zobel et. al. [ZOB93].

SIGNATURE FILES

n-Grams

Before discussing the various forms of signature files, we will describe the *n*-gram concept in greater detail. An *n*-gram is a substring of *n* consecutive characters culled from a string. For instance, the 3-grams of RETRIEVAL are RET, ETR, TRI, RIE, IEV, EVA, and VAL. These *n*-grams are used to index words in the lexicon. In the inverted file system, for example, the word list associated with the 3-gram RET might be RETRIEVAL, RETURN, RETROACTIVE, and PRETTY. In the signature file method, an *n*-gram is hashed to a bit signature of length *F* with *S* on-bits, and *n*-gram signatures are superimposed to generate a *term signature*. This is shown in greater detail in Figure1.

Sequential Signature Files: SSF

The basic signature file method uses term signatures to represent words in a document. In its most basic form, SSF, the *N*-word lexicon is hashed to a signature file of *NF* bits. When the user enters a query, the query term is hashed to its corresponding term signature. It is then compared by simple logical AND to each signature of the signature file, and matches are compared to the query (term) to verify. Since signatures are approximate representations, some lexicon terms may pass the signature file processing phase although they do not match the query. Such terms are called *false drops* and they must be accessed and eliminated by using the actual query n-grams. Therefore, the performance of a signature file method is affected by the number of false drop records (FD). If FD can be estimated accurately, signature file parameters or processing strategy can be adjusted to obtain a better response time [KOC97, KOC99].

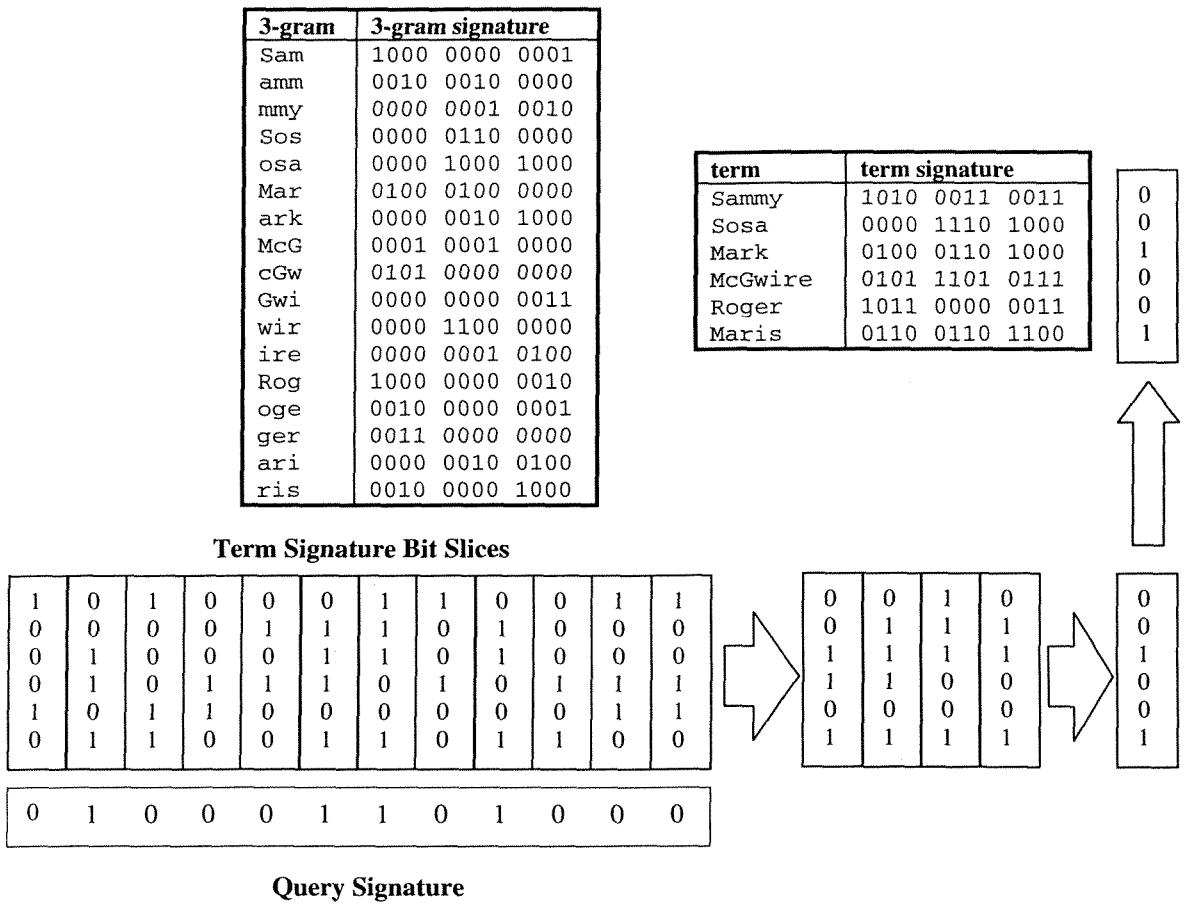

Figure 1. Illustration of Bit-Sliced Signature Files.

Bit-Sliced Signature Files: BSSF

We could cut out a lot of the processing time by utilizing the facts that F is much less than N and that S is much less than F. The signature file can be seen as an N by F matrix. Once we have the query signature, we find the position of each on-bit in the signature. We take the columns (bit slices) corresponding to the on-bit positions and logically AND them all together. Any on-bits in the resulting N-element vector correspond to words that may be matches for the query term. Each of these possible matches is compared to the query to eliminate false matches. This method is called "bit-sliced signature processing" [ROB79].

Figure 1 illustrates the BSSF concept. In this example, F = 12, S = 2, and N = 6. There are 17 separate 3-grams. The lexicon consists of the words "Sammy", "Sosa", "Mark", "McGwire", "Roger", and "Maris". The signatures for the terms are generated by superimposing the signatures of all the 3-grams in the term. The user enters the query term "Mark". A query signature is generated, and the slices corresponding to on-bits in the query signature (the grayed columns in Figure 1) are logically ANDed to generate an answer vector (shown on the far right of Figure 1). The on-bits in the answer vector correspond to possible matches. In this case, "Mark" and "Maris" are the possible matches. Then the false drop resolution process begins to check for false matches. "Maris" is eliminated at this stage.

TEST DATA

We used four text files in the following tests (Table 1). The King James Bible, from Project Gutenberg, has 17,594 unique words. *Ulysses*, by James Joyce, has almost twice as many unique words: 34,035. *lex* is a computer-generated lexicon of 248,969 unique and randomly-chosen words [ZOB98]. Our final index, *all*, combines the three above with a lexicon of Turkish words based on a database of newspaper articles. It has 543,002 unique words. This combined lexicon provides us a large experimentation environment. Due to the agglutinative nature of Turkish words the average size of words in *all* is longer than that of the other lexicons. These four lexicons were used in all tests throughout the paper. All tests were based on a set of 500 queries (words) randomly chosen from the text being tested. The times shown represent the average for 500 queries. Our 3-grams include small- and uppercase letters, digits, and some special characters such as apostrophe.

We used the following conventions throughout our tests:

- 3-grams were used because they are generally thought to give good IR performance [ADA93], and the best ratio of search time to memory required. 2-grams require less space but more time; 4-grams require more space but less time.
- All the lexicons are sorted. Although using unsorted databases might better represent the uses of our system, the inverted file system we will compare ours to works best with sorted lexicons.
- All 500 test queries are *partial queries*. This means that some of the letters are replaced by wildcards. A query must have at least one 3-gram to be considered valid.

All tests were done on a dual processor 180MHz Pentium Pro running Linux 2.0.34.

Table 1: Properties of lexicons

	Bible	*Ulysses*	*lex*	*all*
Size (Kb)	128.9	282.9	2,160.4	4,986.4
Number of words	17,594	34,035	248,969	543,002
Number of 3-grams	7,376	11,196	28,585	44,653
3-grams per word	4.33	5.31	5.68	6.18

PARTIALLY EVALUATED BIT-SLICED SIGNATURE FILES: PBSSF

BSSF in its initial form will eliminate nearly all false matches. But false drop elimination time is negligible for small numbers of false matches. Could we, instead of processing all the relevant bit slices, only process slices until the time it takes to process another slice is greater than the time it would take to resolve the expected number of false matches? The answer is yes. We call this Partially Evaluated Bit Sliced Signature Files, or PBSSF [KOC96a], and we formally define it in the following paragraphs.

We are trying to minimize search time T in a BSSF of size M bytes.

$$T = t_{slice} \cdot i + t_{resolve} \cdot FD(i)$$

$$M = \frac{F \cdot N}{8}$$

where t_{slice} is the time it takes to process a single bit slice, i is the number of bit slices to process, $t_{resolve}$ is the time it takes to resolve a false match (in our case using *regex*), and $FD(i)$ is the expected number of false drops after i bit slices have been processed. t_{slice} and $t_{resolve}$ are experimentally measured; $t_{resolve}$ is around 4 microseconds while t_{slice} depends on N.

The program takes as input the set of terms that appear in a document domain. It separates each term into its n-grams. Each n-gram is used to generate a unique random number seed. S random bit positions are set in a bit string of length F. Then all the bit strings are superimposed (logical OR) to generate a term signature. The set of term signatures, the *signature file*, is bit-wise stored in main memory. When

the user enters a query term, it is hashed to a signature as above. Then the PBSSF search method is implemented as described above. We stop searching when

$$t_{slice} \geq t_{resolve} \cdot (FD(i) - FD(i+1))$$

i.e., when the time it takes to process an additional slice is greater than or equal to the amount of time it would take to resolve the false drops that could be eliminated by processing another slice.

To estimate *FD(i)*, use *fd*, the probability of a false drop after processing *i* bit slices.

$$FD(i) = N \cdot fd_i = N \cdot op^i$$

where *op* is defined to be the ratio of on-bits to bits in the bit matrix. (In the above formula, we are assuming that all matches are false drops, this is the conventional assumption of signature analysis [CHR84].) The *op* value of the signature file can be estimated as:

$$op = D \cdot \frac{S}{F}$$

This is fairly intuitive: *S/F* represents the *op* value of a single *n*-gram and *D* is the average number of *n*-grams in a term. Thus *op* is estimated as the average number of on-bits in a signature. In this formula we assume that each n-gram will set different bit locations in the term signature, i.e., we ignore possible bit overlaps. This is acceptable since terms usually have a small number of unique n-grams and F >> S.

The true *op* value can be measured experimentally by counting the total number of on-bits in the signature file and dividing that number by *N·F*. The data structure we used for bit strings makes this process simple.

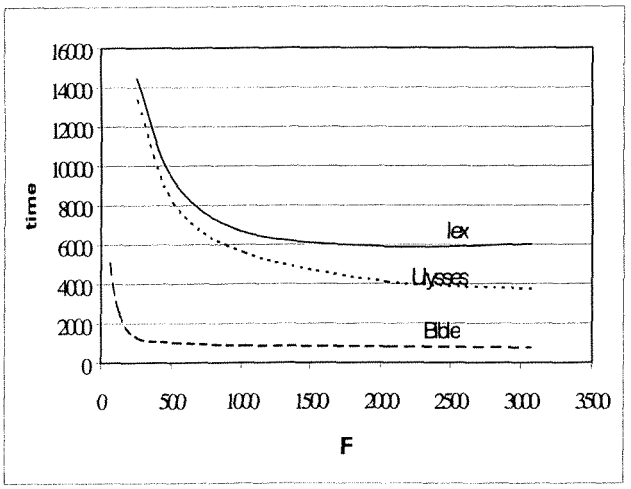

Figure 2. Effects of F size on search time.

Test with PBSSF

Test results show F and search time to be inversely related. An initial doubling of F has a large effect on the reduction of the search time, but repeated doublings of F have less and less effect. Figure 2 shows the relationship between F and search time for all the lexicons except *all*, which is too large to be stored uncompressed in main memory.

Table 2 shows specific numbers for the Bible. Note that after F = 4096, search time levels out and is no longer reduced by increasing F. Size, however, is linearly related to F and increases by whatever factor F is increased by.

Table 2: F and search time for the Bible (S= 1)

	Bible	
F	Time (μsec)	Size (MB)
256	1280	0.56
512	1040	1.13
1024	880	2.25
2048	820	4.50
4096	780	9.00
5120	780	11.26
6144	780	13.51

PARTITIONING

Our previous estimate for the op value does not take all the information we have into account and thus is not a sufficient estimator. A better estimate would take into account the number of terms with d *n*-grams, from d = 1 to $d = d_{max}$.

$$op = \frac{\sum_{d=1}^{d_{max}} \left(d \cdot \frac{S}{F} \right) \cdot D_d}{N}$$

This equation can be logically derived. There are D_1 terms with 1 *n*-gram, and the *op* value of these terms is $1 \cdot S/F$. There are D_2 terms with 2 *n*-grams, and the *op* value of these terms is $2 \cdot S/F$. Add all these up and divide by N and the result is the *op* value of the signature file. This new estimate proves to be an exact match to nine decimal places. It is used to derive the new false drop estimation equation:

$$FD(i) = \frac{N \cdot \sum_{d=1}^{d_{max}} \left(d \cdot \frac{S}{F} \right)^i \cdot D_d}{N} = \sum_{d=1}^{d_{max}} \left(d \cdot \frac{S}{F} \right)^i \cdot D_d$$

It can be shown that with partitioning, S should always be chosen to be 1. Search structure size M does not depend at all on S, so that equation can be ignored. Time T is linearly related to *FD(i)*, which means that T increases as *FD(i)* increases. Therefore we desire *FD(i)* to be as small as possible. *FD(i)* is a function of what we choose S and F to be. Differentiating *FD(i)* with respect to S, we get:

$$\frac{\partial [FD]}{\partial S} = \frac{i}{F} \cdot \sum_{d=1}^{d_{max}} d \cdot \left(d \cdot \frac{S}{F} \right)^{i-1} \cdot D_d$$

It is trivial to show that this is always positive and thus that *FD(i)* is strictly increasing with respect to S. Therefore S should be chosen at its lowest possible value, which is 1.

The expected effect of implementing partitioning is that false drops are more accurately estimated. The program previously may have been underestimating the proper number of slices to process before switching to false drop resolution. This problem will be largely eliminated, resulting in better search times. It is not expected that the times will be noticeably superior to the system without partitioning, however, as the *op* values of our test databases are so small. The difference would be more noticeable in a text with a much larger *op* value.

Tests with Partitioning

While partitioning does provide a more accurate *op* value than before, its effects are not enough to significantly reduce the amount of time needed to perform a search. Its effects might be noticeable if the *op* value is high; i.e. for small values of F. It would be useful, therefore, in systems where space overhead is severely limited. We continue to use it because it is slightly more accurate and the extra calculations are performed outside the search loop so as not to add to processing time at all.

COMPRESSION

Note in Table 2 the large space requirements for even a relatively small text such as the Bible. Since the search structure is stored entirely in main memory, this is a major problem. We could save memory by compressing the signature file in memory. We use a run-length encoding method described by Elias to compress bit slices [ELI75]. Elias' gamma code represents x, the distance between two on-bits, as $\log_2 x + 1$ in unary followed by $x - 2^{\log_2 x}$ in binary. The delta code uses gamma to code $\log_2 x + 1$ and follows it with the same suffix (Table 3). For a run length of less than 15, delta is larger than gamma, but for run lengths greater than 15, delta is always less than gamma. Since we expect large run lengths in general, we use delta encoding. In fact, since the *op* value is generally quite small (less than .01), there is quite a bit of space between on-bits. Delta compression, then, is suited to our purpose.

The fewer on-bits in a bit slice, the shorter the decompression time. When we evaluate a query, we find the shortest bit slices to use in bit slice processing to save as much decompression time as possible. This is done by separating the query term into its n-grams, then using the n-gram signatures to determine which slices are shortest.

Table 3: Examples of Codes

	Coding method	
x	γ	δ
1	1,	1,
2	01,0	010,0
3	01,1	010,1
4	001,00	011,00
5	001,01	011,01
6	001,10	011,10
7	001,11	011,11
8	0001,000	00100,000

The expected result of implementing compression is to greatly reduce the search structure size, but increase search time. With compression enabled, t_{slice} will take into account the amount of time it takes to decompress a bit slice. As a result, fewer bit slices will be processed and there will be more false drops.

Test with Compression

It is interesting to note that while search structures are indeed smaller, search time is not necessarily greater. This can be explained by the way we choose which bit slices to process with and without compression. Without compression, the bit slices to be processed are taken in sequential order based on on-bit positions in the query signature. With compression, the bit slices to be processed are the shortest ones, and thus the ones with the smallest decompression time. We were forced to implement the system this way because choosing the shortest bit slices without compression was inefficient, as was processing bit slices sequentially with compression. The result of this is that there are more false matches when compression is not used, and the time needed to process them offsets the decompression time of the short slices.

Table 5 illustrates the effects of compression based on different selections of F. For F = 1024, the uncompressed database is 110 times bigger than the compressed. For F = 6144, the uncompressed database is a whopping 227 times bigger!

Table 5: The effect of F on search structure size and query evaluation time in *lex*

F	Uncomp. Size (MB)	Compressed Size (MB)	Eval time (μsec) (compressed)
1024	31.9	0.29	13540
2048	63.7	0.48	9400
4096	127.5	0.69	6820
5120	158.9	0.79	6560
6144	190.7	0.84	6140

BLOCKING

In an attempt to save more memory, we can "block" terms together. Terms are normally stored in memory in an array in alphabetical order. With blocking implemented, B term signatures are superimposed to form just one signature. B is called the blocking factor. The size of the uncompressed search structure will then decrease from $NF/8$ byes to $NF/(B \cdot 8)$ bytes. Under compression, the effect of blocking on the size of the search structure is less clear. There will be fewer signatures, but on-bits will be closer together and thus more run lengths will need to be encoded. Blocking may actually have an adverse effect on the size of a compressed bit matrix.

Some modification to our old equations is needed to explain the effect blocking will have on search time. The *op* value will now be expressed in terms of *n*-grams in a signature instead of *n*-grams in a word, and instead of dividing by the number of words, we will divide by the number of signatures in the search structure.

$$op = \frac{\sum_{d=1}^{d_{max}} d \cdot \frac{S}{F} \cdot D_d}{\left\lceil \frac{N}{B} \right\rceil}$$

d now represents the number of *n*-grams in a signature, instead of the number of *n*-grams in a word. *N/B* (rounded up) is the number of signatures in the search structure. The effect of blocking on *op*, then, is to increase it. *FD(i)* is similarly redefined.

$$FD(i) = \frac{N \cdot \sum_{d=1}^{d_{max}} \left(d \cdot \frac{S}{F} \right)^i \cdot D_d}{\left\lceil \frac{N}{B} \right\rceil}$$

Blocking has the effect of increasing the number of expected false drops. It becomes necessary to process more bit slices when blocking is used.

Tests with Blocking

Without compression, the effect of blocking on search structure size is very predictable: it is divided by a factor of *B*. The effect on time is a little less predictable; since t_{slice} now depends on *N/B* (instead of just *N* in the model *sans* blocking) it will be shorter, but the number of false drops increases because signatures are merged. Generally this will cause search time to increase, but in certain cases search time actually decreases when using blocking.

The effect of blocking on the compressed search structure is similar. The difference is that database size will not always decrease; the reason for this is that blocking increases the *op* value, which in turn decreases the run lengths between on-bits, which in turn may (or may not) increase the size of the compressed bit slice. This effect is unpredictable, but lessens as *F* increases.

Table 6 shows the effect of blocking on search structure size and search time on the sorted and compressed *lex* database. When *F* is chosen to be 6144, size always decreases for each increase of *B* and time always increases. When *F* is chosen to be 10000, note that there is a slight size increase from *B* = 4 to *B* = 8, and a significant time decrease from *B* = 1 to *B* = 4. This is due to the effects described above, and tells us that a blocking factor of 4 is best for this case.

Table 6: Effect of blocking factor on compressed search structure size and search time

		lex	
F	B	Size (MB)	Time (μsec)
6144	1	0.84	6260
	4	0.76	6480
	8	0.75	8820
	16	0.73	13740
	32	0.70	23940
10000	1	1.02	7720
	4	0.91	5200
	8	0.92	6780
	16	0.89	10200
	32	0.85	17520

COMPARISON WITH INVERTED FILES

Previous partially specified term search studies include methods such as permuted dictionary mechanism [BRAT82], a variant of the PAT tree-array concept [GON92] called permuterm lexicons [ZOB93], and an inverted file-based string search method that uses the concept of blocking [OWO88]. The blocking method used in the last study is similar to blocking that we use in this paper. The performance of these studies are examined in [ZOB93]. We tested our program against the inverted file method (IF) that is studied in [ZOB93]. IF incorporates the blocking principal of the method defined in [OWO88] and requires considerably less storage than the other methods mentioned above.

Table 7 shows our system's "best" (the *F* and *B* that result in the best search time) result for each database in terms of size of the search structure and search time and compares it to the inverted file result with compression and thresholding as suggested in [ZOB93]. In IF the concept of thresholding implies that when the number of candidate answers falls below a selected percentage of the lexicon size no further index entries are searched and system is switched to false drop elimination process (set to 1% of database size as suggested in [ZOB93]). The thresholding concept is similar to our partial evaluation strategy.

Table 7: PBSSF times vs. inverted file times and PBSSF search structure size vs. inverted file search structure size for each lexicon

	Bible	*Ulysses*	*Lex*	*all*
PBSSF time (μsec)	240	460	5200	20600
IF time (μsec)	240	520	3740	9580
PBSSF size (MB)	0.40	0.50	0.91	1.51
IF size (MB)	0.27	0.57	4.15	9.22

While the inverted file method has a significant advantage in speed, compressed PBSSF has a significant advantage in size. For the *lex* database, for instance, PBSSF is

approximately 2.05 times as slow, but its memory consumption is 4.56 times better. For *all*, PBSSF is 2.15 times as slow while 5.91 times better on memory consumption. The implication is that as database size grows, so does the difference in speed and size between the two systems.

Note that the best time performance of the IF method requires substantial memory--twice as much the original lexicon size (this can be seen by comparing the last row of Table 7 with the first row of Table 1). To decrease the memory requirement of IF we may use blocking. Then the question is by using blocking in IF can we have an IF search structure smaller than that of the PBSSF method and with a time requirement comparable to it.

Our experiments show that IF cannot beat the memory performance of our approach with blocking. We provide the experimental results of *lex* in Figure 3. This figure provides the blocking experiments for *B* size of 32 (left most IF observation), 16, 8, 4, 2, and 1 (right most IF observation). As the figure shows IF cannot beat the memory performance of PBSSF, i.e., by using IF one cannot achieve the memory efficiency of PBSSF with a comparable execution time efficiency. The observations with the other lexicons is the same as this one.

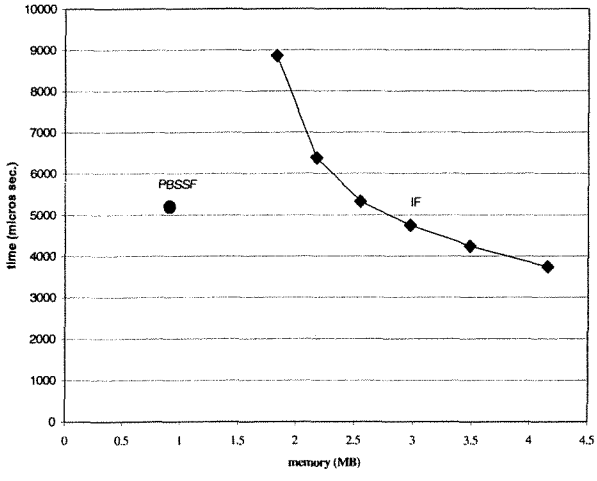

Figure 3. Effects of blocking on performance (lex).

The creation of search structures for *all* is approximately 60 and 600 seconds, for IF and signature-based approaches, respectively. The creation time is proportional to the lexicon size. Since the index structures will be created off-line and used many times both methods can be used in similar environments and the decision will depend on the time-space efficiency concerns of the environment.

CONCLUSIONS

We use the signature file method to search for terms in large lexicons. To optimize efficiency, we use the concepts of the partially evaluated bit-sliced signature file method and memory resident data structures.

Our system uses partitioning, compression, and term blocking in connection with bit-sliced signatures. Index creation using signatures takes more time than that of the inverted file approach; however, the resulting approach provides good response time and is storage-efficient. In the experiments we use partially specified queries, four different lexicons. The experiments show that the signature file and inverted file approaches beat each other in different efficiency aspects. For example, as shown in Table 7, in *all* (our largest lexicon of 543,002 unique words), the IF approach uses 5.91 times more memory but is 2.15 times faster.

ACKNOWLEDGMENTS

We thank Prof. Justin Zobel and his co-workers for making their inverted file programs and the *lex* database publicly available. We also thank Prof. Kemal Oflazer for providing the Turkish text.

REFERENCES

[ADA93] Adams, E. S., Meltzer, A. C. Trigrams as index elements in full text retrieval observations and experimental results. . In *Proceedings of 21st ACM Comp. Science Conference.* (Indianapolis, Indiana, USA). 433-439.

[BRAT82]Bradley, P., Choueka, Y., 1982. Processing truncated terms in document retrieval systems. *Information Processing & Management.* 18, 5, 257-266.

[CHR84] Christodoulakis, S., Faloutsos, C. 1984. Signature files: an access method for documents and its analytical performance evaluation. *ACM Transactions on Information Systems.* 3, 4 (Oct.). 267-288.

[ELI75] Elias, P., 1984. Universal codeword sets and representatives of the integers. *IEEE Trans. on Information Theory*, IT-21. 194-203.

[GON92] Gonnet, G. H., Baeza-Yates, R. A., Snider, T., 1992. New indices for text: PAT trees and PAT arrays. In W. B. Frakes, R. Baeza-Yates, editors, *Information Retrieval Data Structures & Algorithms.* Prentice Hall, Englewood Cliffs, NJ.

[KOC96a] Kocberber, S., 1996. Partial query evaluation for vertically partitioned signature files in very large unformatted databases. Ph.D. dissertation, Dept. of Computer Eng. and Information Science, Bilkent University, Ankara, Turkey (http://www.cs.bilkent.edu.tr/theses.html).

[KOC96b] Kocberber, S., Can, F. 1996. Partial evaluation of queries for bit-sliced signature files. *Information Processing Letters* 60. 305-311.

[KOC97] Kocberber, S., Can, F. 1997. Vertical framing of superimposed signature files using partial evaluation of queries. *Information Processing & Management.* 33, 3, 353-376.

[KOC99] Kocberber, S., Can, F., Patton, J. M., 1999. Optimization of signature file parameters for databases with varying record lengths. *The Computer Journal* (accepted for publication).

[OWO88] Owolabi, O., McGregor D. R. Fast approximate string matching. *Software-Practice and Experience.* 18, 4, 387-393.

[ROB79] Roberts, C. S. 1979. Partial-match retrieval via the method of superimposed codes. In *Proceedings of the IEEE.* 67, 12 (Dec.). 1624-1642.

[SAL89] Salton, G. Automatic Text Processing. 1989. Addison-Wesley, Reading, MA.

[WIT94] Witten, I. H. Moffat, A., and Bell, T. C. 1994. *Managing Gigabytes: Compression and Indexing Documents and Images.* Van Nostrand Reinhold, N.Y.

[ZOB93] Zobel, J., Moffat, A., and Sacks-Davis, R. 1993. Searching large lexicons for partially specified terms using compressed inverted files. In *Proceedings of 19th VLDB Conference.* (Dublin, Ireland). 290-301.

[ZOB98] Zobel, J., Moffat A., Ramamohanarao, K. Inverted files versus signature files for text indexing. *ACM Transaction on Database Systems* (to appear).

Compressed Bit-sliced Signature Files
An Index Structure for Large Lexicons

by
Fazli Can and Ben Carterette
Systems Analysis Department
Miami University
Oxford, Ohio 45056

Working Paper #99-001 04/99