Improving on S-Index: A Hybrid Indexing Scheme for Textbases

Ilias Nitsos
Dept. of Applied Informatics
University of Macedonia
Thessaloniki, Greece
nitsos@uom.gr

Dimitris A. Dervos
Dept. of Information Technology
T.E.I.
Thessaloniki, Greece
dad@it.teithe.gr

Georgios Evangelidis
Dept. of Applied Informatics
University of Macedonia
Thessaloniki, Greece
gevan@uom.gr

Abstract

We present two variations of S-Index and consider their storage utilization efficiency against a 130MB textbase. S-Index is a hybrid-indexing scheme that combines advantages from two popular indexing methods: the inverted file and the signature file. We introduce a new variation of the method and describe the file structure plus the implementation details for both the original as well as for the new variation of S-Index. The performance results obtained are used to compare the two variations. The improved variation of S-Index is measured to utilize less than 5% of the storage utilized by the corresponding indexed textbase corpus, under certain circumstances. The original S-Index involves slightly worse space requirements, but the performance of both variations is comparable to that of the compressed inverted file.

1. Introduction

There are two popular approaches for indexing textbases: Inverted Files and Signature Files or Bitmaps. Both approaches index all the distinct words that belong to a given vocabulary V. Stopwords, i.e., common and function words, such as articles and prepositions, are excluded from the vocabulary and the index.

In the Inverted File index (I-Index) [6] the textbase is divided into blocks. Each distinct word of the vocabulary is associated with an inverted list that contains sorted block identifiers. An address table that maps block identifiers to physical block addresses is used to locate and fetch the actual text from the disk. An efficient and popular structure for storing the vocabulary and the inverted lists is the B+ tree. The inverted lists can be compressed with the use of variable-length codes. The resulting Compressed Inverted File [8, 9] takes up around 5-10% of the size of the textbase being indexed. The reduction in the required disk space for the index is significant, as the above percentage for plain inverted files is 30-60% [9].

Signature Files [3] store signatures (bit-strings) that correspond to words, or blocks of words, of the textbase. A popular variation is the Superimposed Coding - Signature File (SC-SF) [3]. In SC-SF, an F-bit signature is assigned to every word, having m bits set to 1 and the rest set to 0. The textbase is divided into logical blocks, each block containing D distinct words. The signature of a block is generated by OR-ing the signatures of the D distinct words (see Table 1).

Table 1: An SC-SF example having m = 4, F = 16 and D = 4

Word	Signature
free	0010001100100000
text	0000101010010000
database	0000100010010001
example	0000001010010010
Block Signature	0010101110110011

The Signature File for the textbase is the concatenation of all the block signatures. For a single word query the signature of the word is created and all block signatures are searched. A text block is retrieved only if its block signature has at least the same bits set to 1 with the word signature. Next, the text block is scanned for the possibility of the word to still not be contained. A situation where the word signature complies with that of the block but the word does not appear in the latter is

called a false drop [3]. The text blocks that qualify are the ones that are retrieved and found to not correspond to false drops.

A particular instance of SC-SF having m = 1, F = V plus the restriction that distinct words correspond to distinct signatures, is called Bitmap or Exactly Reversible Signature File (ERSF) [1]. Using a minimal perfect hash function (MPHF) [4, 5], or a B+ tree, a unique number in [0,...,V-1] can be assigned to each distinct word of the vocabulary. Then, the signature of the word that corresponds to number k (k in [0,...,V-1]) is a unique V-bit signature having only its k^{th} bit set to 1 and the remaining V-1 bits set to 0. The signature for an entire block is created by OR-ing the unique signatures of the D distinct words of the block. In the end, the block signature has D bits set to 1 and V-D bits set to 0. An evident and less complex way for constructing the signature for a block is to set the D bits, that correspond to the D distinct numbers assigned to each of the D distinct words in the block, to 1 and the remaining V-D bits to 0 (see Table 2 for an example). For a single word query the number k corresponding to the word is retrieved with the help of a B+ tree or an MPHF. If a block signature has its k^{th} bit set to one then the block is retrieved, otherwise it is rejected. By construction, all blocks retrieved qualify for the given query, i.e., there are no false drops.

Table 2: A Bitmap example with V = 16 and D = 4

Word	Word number
free	3
text	9
database	12
example	0
Block Signature	1001000001001000

Both approaches (Inverted Files and Signature Files) allow for fast query processing, but each one has certain advantages and disadvantages. Unlike Signature Files, Inverted Files are not suitable for set oriented processing and multi-term queries [7]. Signature Files, on the other hand, require serial scanning and produce false drops [3]. S-Index, where 'S' stands for signature and 'Index' implies the inverted file index, is a hybrid indexing scheme combining advantages from the two popular indexing approaches presented above.

The structure and construction of two variations of S-Index are described in Section 2. Section 3 focuses on the physical implementation of the variations considered in Section 2. In Section 4 we present experimental results and compare the performance of the two variations of S-Index and the compressed I-Index. In Section 5 we conclude on the subject and comment on the next stages of the research work in question.

2. The S-Index Schema

Two variations of S-Index are presented: the original [2] and a new improved one. From now on, the former will be referred to as S-Index$_1$ and the latter as S-Index$_2$.

S-Index can be represented by a tree structure. Out of a number of alternative tree-structures considered (trees with different fanouts), the scheme achieving the best performance was found to be the binary tree [2]. Section 2.1 describes the construction of the tree and the contents of its nodes for S-Index$_2$. Section 2.2 explains how all the text blocks containing a search word can be retrieved with the help of S-Index$_2$. A full example is provided in Section 2.3. Section 2.4 describes S-Index$_1$, by focusing on the differences of the two variations.

2.1 Signature Insertion

In S-Index$_2$, each distinct, non-common word is mapped on to a unique number in [0,...,V-1], where V is the total number of distinct words in the textbase, through the use of a B+ tree or a MPHF. The textbase is then divided into logical blocks, each containing D distinct words. For every block an ERSF type signature is created. The size of a signature is $M=2^{\lceil \log_2(V) \rceil}$ (the first power of 2 which is equal to or greater than V). This means that a signature has its bits numbered in the [0,...,V-1,V,...,M-1] range. Since only D of the bits [0,...,V-1] can be set to 1 in a signature, bits in [V,...,M-1] are always set to 0 and are reserved for future vocabulary expansion.

Once an ERSF type signature is created for a text block b, it is inserted into the S-Index$_2$ (binary) tree structure. Starting from the root, if the number of 1s in the signature of block b is greater than or equal to the number of 0s, then the signature is stored under the root node, alongside with the corresponding block identifier b, and the algorithm terminates. On the other hand, if the number of 1s is less than the number of 0s, the signature is divided into two equally sized signatures. The first half contains the left M/2 bits and the second half the right M/2 bits of the original signature. The two halves are inserted recursively into the left and right subtrees of the root. The recursive pseudo-code for the insertion of an ERSF type signature of a block in S-Index$_2$ is shown in Figure 1.

Because an S-Index$_2$ tree is a full binary tree, if V is not a power of 2, then some of the nodes exist but are not used. The full binary tree contains $\log_2(M)$ levels of nodes, since this is the number of levels needed, so that it is impossible for a signature to be further divided. A block signature may be divided all the way up to the point where two-bit sub-patterns are produced. The only possible combinations for such two-bit patterns are 01 and 10. This is because 00 is not stored and 11 implies that the corresponding signature would have been appended as a four bit pattern, one level up from the current one. By convention, the root level is at level 0 and the leaf level is at level $\log_2(M) - 1$.

```
PROCEDURE Insert_Signature_Into_Subtree(signature, node)
BEGIN
   number_of_1s = number of 1s in signature;
   number_of_0s = number of 0s in signature;
   IF number_of_1s = 0 THEN exit();

   IF (number_of_1s ≥ number_of_0s in the signature)
   THEN
   BEGIN
      append a new record under the current node;
      exit from this procedure;
   END
   ELSE
   BEGIN
      signature_L = left half of signature;
      signature_R = right half of signature;
      node_L = left child of node;
      node_R = right child of node;
      CALL Insert_Signature_Into_Subtree(signature_L, node_L);
      CALL Insert_Signature_Into_Subtree(signature_R, node_R);
   END
END.

CALL
Insert_Signature_Into_Subtree(ERSF_Type_Signature_For_Block,
root);
```

Figure 1: Pseudo-code for inserting a signature into S-Index$_2$

2.2 Single term query processing with S-Index$_2$

Block signatures appended under the root node utilize the whole of the textbase vocabulary. The left child of the root utilizes the left (lower) half of the vocabulary [0,...,M/2-1], because only the left (lower) part of an ERSF signature can be stored under that node. Similarly, the right child of the root registers information on blocks containing words with codes in [M/2,...,V-1], because only the right part of an ERSF signature can be stored there, and so on.

When attempting to find all the blocks that qualify for a given single term query, one has to start from the root node and follow a single path down to the leaf level. Along this path, all signatures under each one of the nodes encountered are fetched and examined. If the appropriate bit is set to 1, then the block number is retrieved. The block numbers retrieved are then mapped to addresses on the disk, with the help of an address table constructed during the partitioning of the textbase into logical blocks. The actual text may then be fetched from the disk, by following the corresponding address pointers.

2.3 Detailed example

As an example, let us consider the following text: "This is an example for a small text database with common words. Common words in the text are not indexed." The size of the vocabulary in this textbase is V=7, so M=8 (1 bit is wasted). The mapping of the words indexed is presented below.

example	0
small	1
text	2
database	3
common	4
words	5
indexed	6

The textbase is divided into logical blocks, each containing D=3 distinct words:

Block number	Block context	Block signature
0	This is an example for a small text	S_0 = 11100000
1	database with common words.	S_1 = 00011100
2	Common words in the text	S_2 = 00101100
3	are not indexed	S_3 = 00000010

The empty and populated S-Index$_2$ structures appear in Figure 2 and Figure 3, respectively.

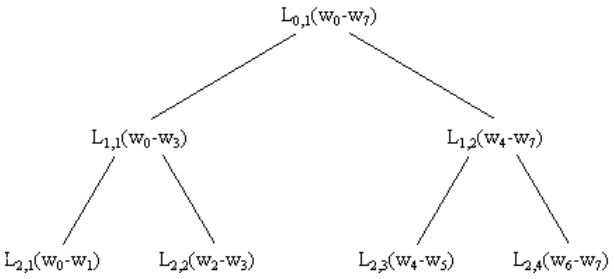

Figure 2: Empty S-Index$_2$ structure

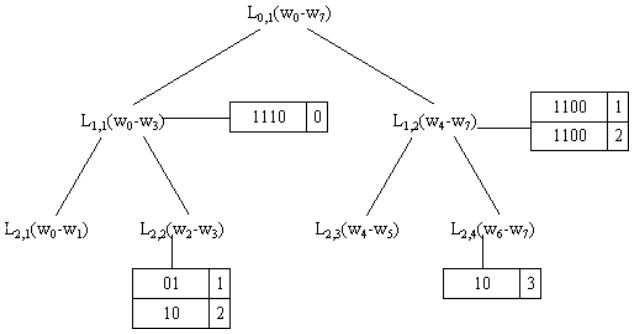

Figure 3: Populated S-Index$_2$ structure

Let's see how block signature S_1 is inserted into the index. S_1 cannot be placed under the root because it has three 1s and five 0s. Thus, it is divided into two signatures S_{1_L} (0001), and S_{1_R} (1100). S_{1_L} cannot be placed under node $L_{1,1}$ because it contains one 1 and three 0s. S_{1_L} is therefore divided into $S_{1\_L\_L}$ (00) and $S_{1\_L\_R}$ (01). $S_{1\_L\_L}$ is not stored under node $L_{2,1}$ because it contains no 1s.

$S_{1\_L\_R}$ is stored under node $L_{2,2}$, alongside with its block identifier, since it contains one 1 and one 0. In a similar way, S_{1_R} and block number 1 are appended to the list under node $L_{1,2}$, because half of the signature bits are set to 1.

In the course of processing the single term query "find all the occurrences of the word text in the textbase", first the word text has to be mapped to its corresponding code, which is number 2. This is achieved with the help of a B+ tree or a MPHF. Next, we consider the root of the S-Index$_2$ tree. For the example in question, no records are stored under the root node, thus the search algorithm chooses a child node to continue. The latter is node $L_{1,1}$ and is determined by considering the word code (2, in this case). All of the signature records stored under the given node are retrieved and checked. When the appropriate bit position (here, bit position 3) is set to 1, the corresponding block identifier is retrieved. In this case, block number 0 is retrieved. The next child node to be considered is $L_{2,2}$. Bit position 1 of all the signatures stored under this node is checked. The algorithm retrieves block number 2 from node $L_{2,2}$ and terminates because a leaf node has been reached. It is now clear that the word text appears in blocks 0 and 2. This can be confirmed by scanning the corresponding textbase blocks. Figure 4 illustrates the main points of the example considered.

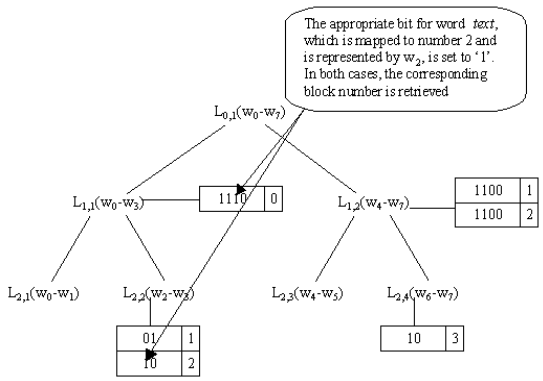

Figure 4: Answering a single term query on text

2.4 S-Index$_1$: Differences and Similarities with S-Index$_2$

We now present S-Index$_1$. We clarify its details by focusing on the signature insertion and the single-term query processing algorithms.

Signature insertion. The signature insertion algorithm remains the same. The only difference is that a record is stored under a node only when the number of 1s in a signature is greater than the number of 0s. This has the following consequences: Level $\log_2(M)$-1 now stores the signature pattern 11. Patterns 01 and 10 cannot be stored under the nodes in level $\log_2(M)$-1, because the number of 1s is equal to the number of 0s. Thus, a new level has to be created, level $\log_2(M)$, in order to store the one-bit patterns generated by the division of the above two-bit patterns. It is obvious, that there is no need to store signature patters under the nodes at levels $\log_2(M)$-1 and $\log_2(M)$, because only one pattern corresponds to each of them, namely, pattern 11 for level $\log_2(M)$-1 and pattern 1 for level $\log_2(M)$.

Single term query processing. The only difference in the search algorithm is that no bit comparison needs to be made for records stored under nodes in the last (lower) two levels of the tree. It is by construction that all block identifiers stored under a node at these levels correspond to blocks known to qualify for the query in question.

Detailed example. The textbase used for this example is the same as the one used in subsection 2.3. It is partitioned into the same logical blocks and the same ERSF type signatures are created. The corresponding empty and populated tree structures are shown in Figure 5 and Figure 6, respectively. Figure 7 summarizes on the search algorithm for S-Index$_1$.

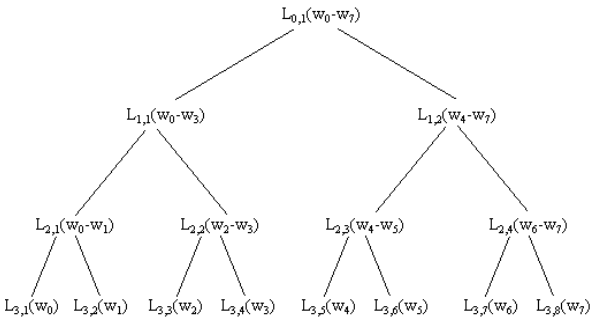

Figure 5: Empty S-Index1. Note that a new level (level 3) has been created

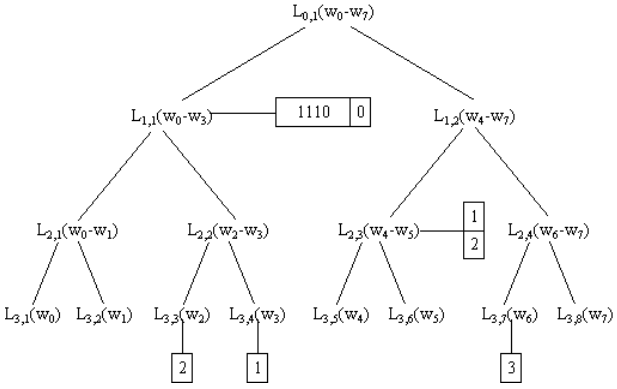

Figure 6: Populated S-Index1. Note that no signatures are stored in the last two levels

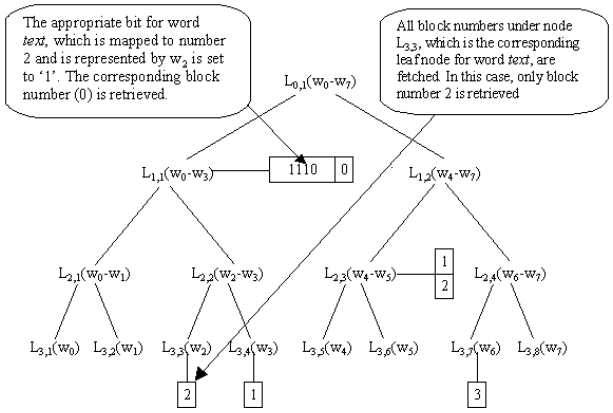

Figure 7: Answering a single term query on text. Note that no bit comparison is necessary for the lower two levels

3. Implementation

This section covers the implementation details for S-Index$_1$ and S-Index$_2$, and focuses on the latter. As shown in Figure 3, S-Index$_2$ consists of two main parts: (a) the skeleton which is a full binary tree containing the nodes, and (b) under each node a list of records, each being of the (block identifier, block signature) type. In the following we describe the way this structure is organized on disk.

Root level (level 0). All records under the root node are stored in a linked list on the disk. A typical root record stores a signature of size M and a block number. The signature utilizes $\lceil M/8 \rceil$ bytes on the disk and the text block number is stored as a two-byte integer. A pointer to the first record of the root's record list on the disk is stored in the first row of a look-up table containing $\log_2(M)$ rows. A traversal of the list retrieves the remaining records.

Intermediate level (level i, where $0 < i < \log_2(M) - 1$). All records, in all lists, under all nodes in level i, are stored on the disk in a combined linked list - a structure that consists of many individual linked lists. A typical record, in the list under any node in level i, consists of a $M/2^i$ bit signature, a text block number and a pointer to the next record in the list. The signature utilizes $\lceil M/2^{i+3} \rceil$ bytes on the disk. In addition, two bytes are used for the block number and four are reserved for the next record pointer.

The combined linked list is constructed with the help of the following algorithm: At first, space for 2^i records is allocated sequentially on the disk. Each one of these empty records is the head record to the list of each of the 2^i nodes in level i, and initially has the next record pointer and the text block identifier field set to NULL. In the $(i+1)^{th}$ row of the lookup table, a pointer to the first record of the combined linked list is stored. Once the head record of a list has been used, new records are appended right after it.

In the example below we demonstrate the implementation of the combined linked list for the third level (level 2) of an S-Index$_2$ tree having M = 16. The size of a signature for level 2 is $\lceil M/2^i \rceil = \lceil 16/2^2 \rceil = 4$ bits. Figure 8 illustrates the four nodes at level 2, and Figure 9 illustrates the corresponding combined linked list of records.

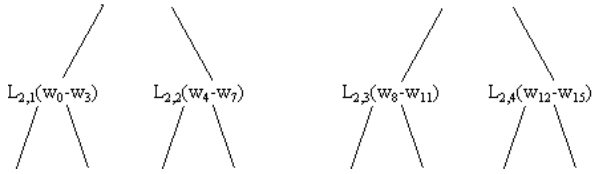

Figure 8: The empty structure for level 2 of S-Index$_2$ when M = 16

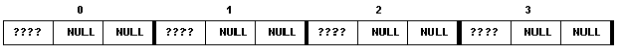

Figure 9: The empty combined linked list

Record 0 is the empty head record for node $L_{2,1}$, record 1 is the empty head record for node $L_{2,2}$ and so on. Space for four empty head records has been allocated serially on the disk.

After four insertions, the S-Index$_2$ schema and the corresponding combined linked list at level 2 are shown in Figure 10 and Figure 11, respectively.

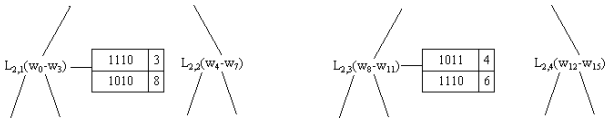

Figure 10: The tree structure after four record insertions

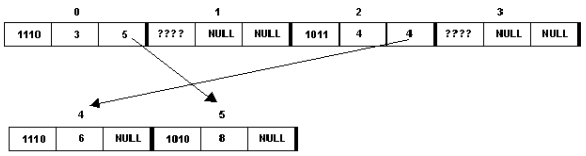

Figure 11: The combined linked list after four record insertions

Suppose now that record (1110,9) is to be inserted into the list under node $L_{2,3}$. Figure 12 and Figure 13 illustrate the updated tree and combined linked list structures for this case.

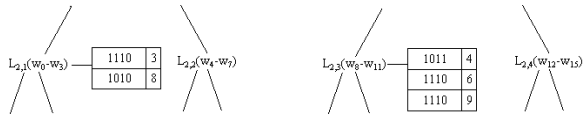

Figure 12: The tree after the insertion of (1110, 9) under $L_{2,3}$

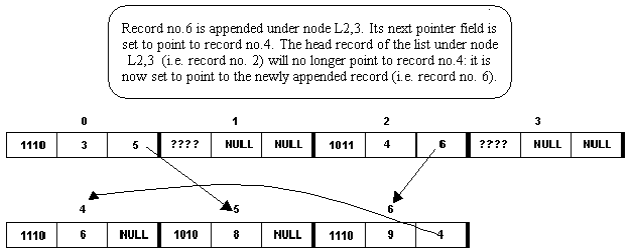

Figure 13: The combined linked list after the insertion of (1110, 9) under $L_{2,3}$

In order to traverse the entire list under node j of level i ($L_{i,j}$ where $0 < i < \log_2(M)-1$ and $1 \le j \le 2^i$) we need to: (a) go to the combined linked list with the help of the $(i+1)^{th}$ row of the look-up table, (b) jump to the j^{th} record of the combined linked list, which is the head record to the list under node $L_{i,j}$, and (c) retrieve the whole list by following the next record pointer, until a NULL value is encountered.

Leaf level (level $\log_2(M)-1$). The leaf level is stored with the help of a combined linked list, like any intermediate level. The differentiation has to do with the signatures it registers. In the case of S-Index$_2$, the possible combinations for the two bit signatures in the leaf nodes are 01 and 10. Thus, for each one node we maintain two record lists, one for each one of the two (possible) 2-bit signature patterns. Evidently, each one record of the two lists in question stores a (block identifier, next record pointer) pair.

From the discussion above, it follows that the binary tree (backbone) structure is neither saved on the disk nor is maintained in main memory. The only structure that needs to be stored is the look-up table that directs to the combined linked lists and allows for efficient retrieval of the records stored under the S-Index$_2$ nodes.

The implementation of S-Index$_1$ is along the same lines. The difference lies in the two lower levels of S-Index$_1$. As explained in subsection 2.4, no signature patterns have to be stored under the nodes of the lower two levels. This means that a typical record in the combined linked lists of the two last levels of the tree is of the (block identifier, next record pointer) type. At its lowest (leaves) level, S-Index$_1$ turns into an I-Index type structure. It is in this respect that S-Index is said to comprise a hybrid-indexing scheme.

4. Experimental Results

In order to compare the performance of S-Index$_1$ with that of S-Index$_2$ and the I-Index, a series of experiments were conducted. We developed a version of S-Index that can be tuned to perform either like S-Index$_1$ or like S-Index$_2$. For an I-Index environment, we used the MG system[1] [9], which utilizes compressed inverted files for indexing textbases. The textbase used for the experiments is a 130 MB (synthetic) document collection that was generated with FINNEGAN[2] [9]. Its profile is outlined in Table 3. FINNEGAN is a textbase generator that can create synthetic textbases of any size with statistical properties analogous to those of real text. The experimental results obtained are used to compare the two variations of S-Index with regard to their storage utilization efficiency. S-Index has been found to achieve a performance comparable to that of file inversion.

Table 3: The textbase profile

Collection Size	130 MB
Filter Size (number of common and function words)	598
Vocabulary Size (number of distinct words)	128727

In our implementation of S-Index, the mapping of the distinct words to unique code numbers is achieved with the help of a B+ tree. The numbers are assigned to the words sequentially. This means, that the number code assigned to first distinct word of the textbase is 0, the number assigned to the second distinct word is 1, and so on. The ERSF type signatures generated for each block are 131072 ($= 2^{17}$) bits long, because the vocabulary size was measured to be 128727 words. The binary positions utilized during block signature construction range from bit position 0 to bit position 128726. The remaining 2345 bits may be used for future vocabulary expansion.

Figure 14 presents the results obtained by considering several blocking factor (D) values for S-Index$_1$ and S-Index$_2$, and includes the curve of the compressed I-Index scheme [9]. The size of the S-Index$_1$ and S-Index$_2$ indexes decreases as the value of the blocking factor (D) increases. S-Index$_1$ is seen to require more space on disk than S-Index$_2$, for the same D values. Moreover, for a particular range of D values, S-Index requires much less space on disk, than I-Index. For the textbase considered, if D = 4500 then S-Index$_2$ and I-Index produced indexes of the same size. S-Index$_1$ and I-Index produced indexes of about the same size for a value of D around 5500. Nevertheless, upon increasing the value of D, the S-Index

[1] MG is available via ftp from ftp://munnari.oz.au/pub/mg.
[2] FINNEGAN is available via ftp from ftp://munnari.oz.au/pub/finnegan.

variations produced more compact indexes. For D = 12000, S-Index$_2$ was measured to be only 4.28% the size of the indexed textbase and S-Index$_1$ 5.21%. For S-Index$_2$, this implies an improvement of about 18% over S-Index$_1$, and 57% over the I-Index.

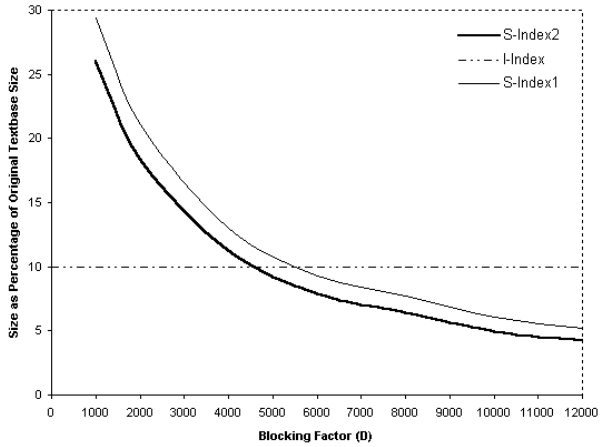

Figure 14: Disk space requirements for S-Index$_1$, S-Index$_2$ and I-Index

The cost for increasing the size of the blocking factor in S-Index is an increase in the size of the logical blocks, into which the textbase is partitioned. The curve in

Figure 15 illustrates the impact that different values of D have on the average text block size. Large text blocks slow down the query processing speed, because each block has to be scanned, so that the exact position of the word in question is found. This means that the choice of D is important in order to achieve the desirable performance with regard to disk space requirements and query processing efficiency. S-Index$_2$ appears to be a better choice over S-Index$_1$, since it has been measured to produce more compact indexes.

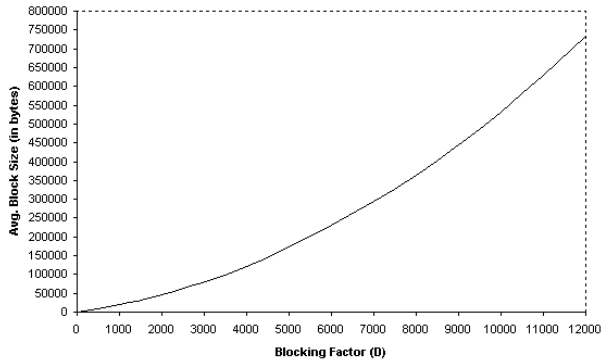

Figure 15: The average block size is affected by the blocking factor

5. Conclusion

In this paper we presented in detail the schemata and the implementation of the basic and of an improved variation of S-Index. The storage utilization efficiency of the two S-Index variations was considered next to a 130 MB textbase. The experimental results obtained indicate that the improved variation may be configured to utilize an index size that is less than 5% the size of the indexed textbase corpus. This in turn implies performance comparable to that of the compressed inverted index. For S-Index, the decrease in the index size is achieved at the cost of increasing the average logical text block size. The latter implies a decrease in query processing efficiency.

This study emphasizes on measuring the performance of S-Index with regard to storage utilization efficiency. In the future stages of our research, we intent to focus on the S-Index query processing efficiency. It is worth noting that the scheme allows for further improvement with regard to storage utilization efficiency: the list of records stored under the leaf nodes are of the same type as those of the inverted index. In this respect, S-Index is expected to benefit from implementing compression, analogous to that of the self-indexing inverted files, reported to reduce the size of each one record down to one byte [8].

References

1. Dervos D., Linardis P. and Manolopoulos Y.: "Perfect Encoding: A Signature Method for Text Retrieval", Proc. International Workshop on Advances in Databases and Information Systems (ADBIS), pp.176-182, Moscow, Sept. 1996.
2. Dervos D., Linardis P. and Manolopoulos Y.: "S-Index: A Hybrid Structure for Text Retrieval", Proc. First East European Symposium on Advances in Databases and Information Systems (ADBIS), pp.204-209, St. Petersburg, Sept. 1997.
3. Faloutsos C.: "Signature Files: Design and performance comparison of some signature extraction methods", Proc. ACM SIGMOD, pp.63-82, 1985.
4. Fox E.A., Chen Q.F., DAOUD A.M. and Heath L.S.: "Order-Preserving Minimal Perfect Hash Functions and Information Retrieval", ACM Transactions on Information Systems, Vol.9, No.3, pp.281-308, July 1991.
5. Fox E.A., Chen Q.F. and Heath L.S.: "A Faster Algorithm for Constructing Minimal Perfect Hash Functions", Proc. ACM SIGIR Conference, pp.266-273, Copenhagen, June 1992.
6. Harman D., Fox E., Baeza-Yates R.A., and Lee W.: "Inverted Files", in Information Retrieval: Data Structures and Algorithms, by Frakes W. and Baeza-Yates R. (Eds.), Prentice Hall, pp.28-43, 1992.
7. Jagadisch H. and Faloutsos C.: "Hybrid Index Organizations for Text Databases", Proceedings of The Extending Database Technology Conference (EDBT), pp.310-327, March 1992.

8. Moffat A. and Zobel J.: "Self-Indexing Inverted Files for Fast Text Retrieval", ACM Transactions on Information Systems, Vol.14, No.4, pp.349-379, Oct. 1996.

9. Zobel J., Moffat A. and Ramamohanarao K.: "Inverted Files Versus Signature Files for Text Indexing", ACM Transactions on Database Systems, Vol.23, No.4, pp.453-490, Dec. 1998.