The following paper was originally published in the
Proceedings of the USENIX Symposium on Internet Technologies and Systems
Monterey, California, December 1997

SASE: Implementation of a Compressed Text Search Engine

Srinidhi Varadarajan and Tzi-cker Chiueh
State University of New York

For more information about USENIX Association contact:

1. Phone: 510 528-8649
2. FAX: 510 548-5738
3. Email: office@usenix.org
4. WWW URL: http://www.usenix.org/

SASE: Implementation of a Compressed Text Search Engine

Srinidhi Varadarajan Tzi-cker Chiueh

Department of Computer Science
State University of New York
Stony Brook, NY 11794-4400

(srinidhi, chiueh)@cs.sunysb.edu

http://www.ecsl.sunysb.edu/RFCSearch.html

Abstract

Keyword based search engines are the basic building block of text retrieval systems. Higher level systems like *content sensitive* search engines and knowledge-based systems still rely on keyword search as the underlying text retrieval mechanism. With the explosive growth in content, Internet and Intranet information repositories require efficient mechanisms to store as well as index data. In this paper we discuss the implementation of the **Shrink and Search Engine** (*SASE*) framework which unites text compression and indexing to maximize keyword search performance while reducing storage cost. *SASE* features the novel capability of being able to directly search through compressed text without explicit decompression. The implementation includes a search server architecture, which can be accessed from a Java front-end to perform keyword search on the Internet.

The performance results show that the compression efficiency of *SASE* is within 7-17% of GZIP one of the best lossless compression schemes. The sum of the compressed file size and the inverted indices is only between 55-76% of the original database while the search performance is comparable to a fully inverted index. The framework allows a flexible trade-off between search performance and storage requirements for the search indices.

1. Introduction

Efficient search engines are the basic building block of information retrieval. Content sensitive engines like Lycos and Yahoo still rely on keyword search as their underlying search mechanism. Furthermore, with growth in corporate intranet information repositories, efficient mechanisms are needed for information storage and retrieval.

In this paper we propose a scheme to maximize keyword search performance while reducing storage cost. The basic idea behind the proposed framework called the **Shrink and Search Engine** (*SASE*), is to use the commonality between dictionary coding and inverted indexing to unite compression and text retrieval into a common framework. The result is a search engine that is efficient both in terms of raw speed as well as storage requirement, and has the capability of searching directly through compressed text.

This paper is organized as follows. Section 2 describes the basic idea behind *SASE*. In section 3 we discuss the implementation issues and our Internet *SASE* Server architecture. Section 4 reports the results of a performance analysis of our system. In section 0, we present related work in the area. Section 6 concludes the paper with a report on the major results and future work in the area

2. Basic Algorithm

The common approach to fast indexing uses a structure called the *inverted index*. An inverted index records the location of each word in the database. When a user enters a query word, the inverted index is consulted to get occurrence list of the word. Typically the inverted index is maintained as a dictionary with a linked list of occurrence pointers associated with each word. The dictionary is organized as a hash table for faster keyword search.

A significant characteristic of textual data is the high degree of inherent redundancy in it. Text compression reduces source redundancy by substituting repetitive patterns with shorter numerical identifiers. Text compression can be done by variable bit length statistical schemes like Huffmann coding or dictionary based schemes like LZW, which substitute identical character strings with dictionary identifiers representing the pattern. Our observation here is, that both inverted indexing and dictionary based text compression require a dictionary. Hence one can reuse the dictionary from the inverted index for dictionary coding uniting

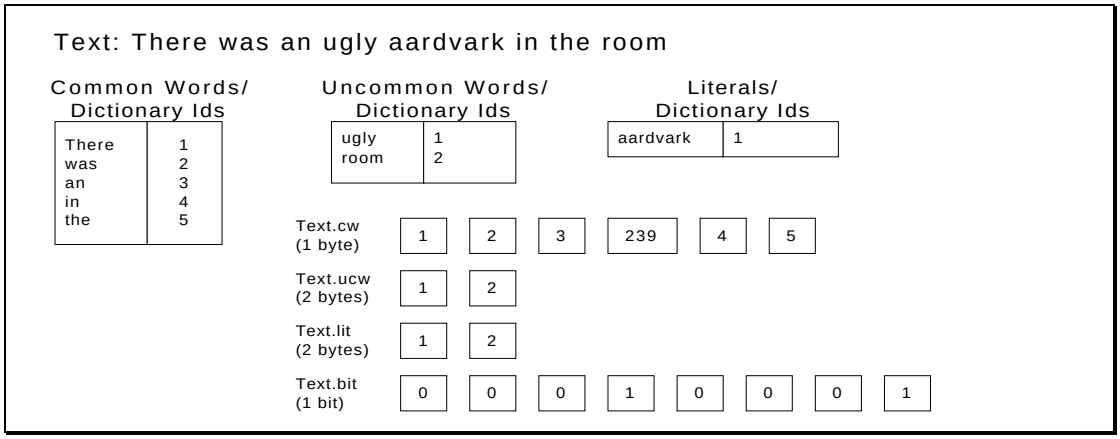

Figure 1: The compressed text representation of an example string. The literal word "aardvark" is represented by a reserved code 239 in the common word file

compression and pattern matching into a common framework.

Dictionary based compression can be done at several levels of token granularity. In our united compression/pattern matching framework, we use a *word* as the basic dictionary element. A word is any pattern punctuated by white-space characters. The advantage of this approach is that it integrates the requirements of word based pattern matching and compression. The drawback is that the compression efficiency is not as high as that obtained from dictionary schemes like Lempel-Ziv which use arbitrary string tokens.

Text compression is performed in *SASE* by substituting words with their numerical representation called *lexical codes*. To improve the utilization efficiency of the available lexical code space, we use a technique similar to Huffmann coding at the byte level. The set of words in a database is partitioned into three groups viz. *common words*, *uncommon words* and *literals*. Common words occur more frequently than uncommon words, which in turn occur more frequently than literals. The classification is done on the basis of the *compression benefit factor* (CBF) of a word, which is defined as the product of the length of the word and its occurrence count. This partitioning is done off-line since the target applications for this scheme are mainly read-only databases. In the common word dictionary, words are represented by a 1 byte code. The uncommon word and literal dictionaries use a 2 byte code. Our experiments show that common words occur more than 50% of the time and greatly benefit from their smaller representation.

2.1 Compression and Decompression

In order to compress a text database, the database is first scanned to determine the list of unique words sorted by their compression benefit factors. The first 256 words are put in the common word dictionary and the next 64K words are put in the uncommon word dictionary. The second pass is done during the compression phase where each word in the database is converted to its dictionary id. In this pass literals are identified and literal dictionaries are created on demand. This scheme allows us to share the common and uncommon word lists across multiple *similar* databases. Compression on such databases would need only one pass.

The compressed representation of a text file consists of the following four files:

1. **.cw* : A file of common-word dictionary IDs, each of which is represented as a 1-byte codeword indexing into the common word dictionary. There are some exceptions. Ten of the 256 1-byte codewords are used as special flags to indicate that the next word is a literal whose 2 byte code is in the literal file. Some other codes are used to optimize capitalization and for run-length-encoded tokens, as explained in Section 3.1
2. **.ucw*: A file of uncommon-word dictionary IDs, each of which is represented as a 2-byte codeword indexing to the uncommon word dictionary.

3. **.lit*: A file of literals, each of which is represented by a 2-byte codeword indexing to the literal dictionary.
4. **.bit*: A bitmap file in which each bit represents a word in the text database and indicates whether it is a common word/literal or an uncommon word.

Fig. 1 shows the compressed representation of the string "*There was an ugly aardvark in the room*". The words *there, was, an, in* and *the* are assumed to be common words and are assigned the dictionary ids 1, 2, 3, 4 and 5 in the common word dictionary. Similarly *ugly* and *room* are uncommon words and are assigned the ids 1 and 2 in the uncommon word dictionary, whereas the word *aardvark* is a literal and is assigned the code 1 in the literal dictionary 1. In the compressed representation of the string, the bitmap file is used to direct the decompression engine to go to either the compressed common word file or the uncommon word file. To get the next code from the literal file we indicate that the next word is a common word and then use a special code in the common word file to further direct the decompression engine to get the next word from the literal file. Codes 239 to 249 are reserved in the common word file to direct decompression to literal dictionaries 1 to 10.

While this scheme is roughly modeled on the lines of Huffmann coding, it has two distinct advantages over Huffmann coding. First the code space is used more efficiently since individual dictionary ids do not have to satisfy the unique prefix property. Secondly codes of different length for different dictionaries are maintained in independent files. A bitmap file consisting of 0s and 1's is used to direct the decompression scheme. A 0 indicates that the next word is in the common word file whereas a 1 indicates that the next word is in the uncommon word file. A reserved code in the common word file may further indirect the decompression to read the next code from the literal files. This scheme can be considered an instance of Huffmann coding with a 1 bit prefix.

The number of unique words found in *normal* databases (stories, newspaper articles etc.) is quite small. However, technical databases tend to have a very large vocabulary, particularly when they contain computer program code or ASCII art. To accommodate these words, *SASE* reserves code space in the common word file to support up to 10 literal word dictionaries of 64K words each for a total of 704K words. More codes may be reserved to support larger databases at a small penalty in compression ratio due to the increased number of reserved tokens.

2.2 Indexing and Searching

In a full inverted index structure, each dictionary entry consists of a linked list, which records the positions of all instances of the word. When the user enters a keyword, the linked list is followed to obtain all the occurrences of a keyword. While this scheme has very fast search times, the space complexity of generic full inverted indexing schemes is quite large. The size of the inverted index has been reported to range between 50%-300% of the size of the original database[FALO85].

SASE solves this problem by using an indexed approach. The text database is partitioned into blocks by partitioning the bitmap file into equal sized chunks. The pointers in the linked list are block identifiers. Note the partitioning is in terms of bits in the bitmap file, for example a 4KB block size contains 4K*8 = 32K words. The first occurrence of a keyword in each block is recorded irrespective of the number of occurrences of the keyword in the block. In order to reach the other occurrences, a linear search is performed on the block. This scheme allows a flexible trade-off between speed and storage requirement. With a smaller block size, it takes less time to search through it, whereas the space requirement increases since there are more block pointers. Conversely, a larger block size requires less block pointers whereas the time required for searching is larger.

In the indexed approach taken by *SASE*, we need to perform a linear search in a block to find other instances of a keyword. A naïve implementation would decompress the compressed text and perform string comparisons between the query word and the decompressed text. Since *SASE* applies dictionary coding in its compression scheme, it is possible to **search directly through the compressed text** without explicit decompression. The query keyword is first converted into its dictionary id and directly compared against the dictionary id's in the compressed text. When an instance of the keyword is found, the location in the compressed files is marked. Future searches can begin from this location. Search within a block is terminated when the number of instances of the keyword found matches the count field associated with a block, which maintains the total number of occurrences of

the keyword. This scheme is much faster than any string comparison based indexing scheme since we only need to perform fixed length numeric comparisons as opposed to variable length string comparisons.

Boolean queries can be performed by AND/OR operations on the linked lists associated with the query keywords. The resultant list formed by the applying the Boolean expression on the linked lists is then searched.

The block size of the inverted index plays a critical role in the performance of *SASE*. An optimization that can be performed here is to use different block sizes for different words. *SASE* implements a fully indexed **dynamic index cache** to reuse results from previous searches. A separate dictionary is used which caches every occurrence of the most frequently/most recently accessed words. When a keyword is searched, the search results are posted to the index cache. Since *SASE* supports next occurrence type of queries, it is possible to have incompletely filled entries in the index cache. These incompletely filled entries are filled when a user query accesses all occurrences of a keyword. The index cache is consulted to see if it can satisfy a request before beginning a search using the inverted index.

2.3 Approximate Search

For approximate searching, the set of uncommon words and literals are statically organized in a **Vantage Point** (VP) [YIAN92][CHIU94] tree. The user specifies the desired maximum number of errors between the query word and his results. We can then traverse the VP tree to get a set of words that fall within the allowable error range.

The set of allowable branches is determined by comparing the query word against the interior nodes of the tree. The remaining branches are pruned since we know that none of their leaf nodes can contribute to the query. Although this scheme performs considerably better than a linear search through the dictionary, the number of comparisons is still high. An interesting observation here is that word lengths are finite and discrete. Hence, we can build multiple VP trees, one for each length. When the user enters a query, the set of allowable VP trees is determined from the length of the query word and the desired maximum number of errors. These VP trees are then searched to get the set of allowable words. Experiments on this scheme show that we need to compare against 4-8% of the words in the dictionary to get the set of allowable words. After the allowable set of words has been determined, we search the database for each word in the set.

3. Implementation

The ITCI compression/search engine has been implemented in C running on a UNIX platform. It is consists of a (i) compression and decompression engine and (ii) a search engine. In our current implementation of *SASE*, we have built a communication subsystem around the search engine to allow searches from the World Wide Web using a Java front-end. In this section we discuss implementation details of the various sub-systems within *SASE*.

3.1 Compression Engine

Before we begin compression, we need to collect statistics to determine the word breakup into common words; uncommon words and literals based on the compression benefit factor. Once these statistics are collected, the compression engine builds up a hash table of common words and uncommon words. Literal hash tables are created *on demand* whenever they are encountered in the text. In this phase we also build an inverted index for uncommon words. Literal inverted indices are created on demand.

After the indices have been built, a parser parses the input token stream to extract words from it. Words are defined as a stream of alphanumeric characters delimited by white space tokens. Several optimizations are performed in this phase.

1. In a stream of natural text, a space character follows each word. It would be wasteful to store a token to represent the space. *SASE* assumes implicitly that each word is followed by a space. Absence of space is encoded with a reserved common word DELETE_SPACE flag, which precedes a token not followed by a space. This allows for better compression under the common case that each word is followed by a space

2. In a text database both normal and capitalized versions of a word can occur. Typically each sentence begins with a word whose first character is capitalized. Multiple versions of the same word take up additional dictionary space. For instance *version*, *Version* and *VERSION*

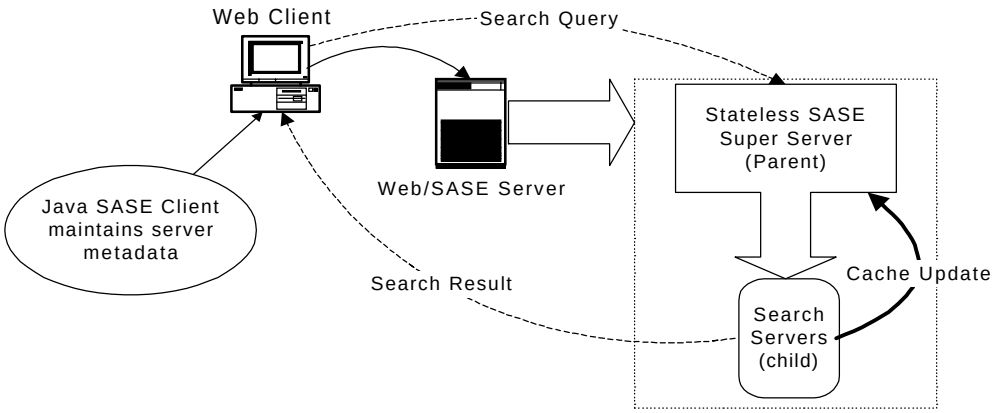

Figure 2 : Execution of a typical search query from then Web. The Java client sends in its query to the *SASE* super server, which forks a copy of itself to perform the search. The child process returns the result to the client along with server metadata. The child also performs a cache update on the super server.

would appear as three independent tokens. To prevent this, we precede capitalized words with a reserved common word code indicating the type of capitalization; i.e. either the first letter or the entire word is capitalized.

3. In typical usage, a sentence is ended by a period after the last word in the sentence with no intermediate space between the period and the word. If we follow the above optimizations, at the end of a typical sentence we would end with a DELETE_SPACE code, followed by a code for the last word and a code for the period character. To optimize this case, we special case periods into normal periods and end of sentence periods. Tokens followed by an end of sentence period do not have a space between them by default. This saves us the byte required for the DELETE_SPACE code. This optimization is also used for commas and semicolons.

4. Many documents have repeating sequences of white space or punctuation characters for typesetting purposes or ASCII art. These tokens take up a lot of space in a dictionary since there are independent tokens for sequences of each length, for example the following sequences ----- and --------- used in ASCII tables represent two tokens in the inverted index. In most cases, these tokens are never searched. To optimize this, we perform Run length encoding (RLE) on space tokens and punctuation characters. Contrary to the 1 byte representation used in the common word file, a run length encoded token uses up 3 bytes in the common word file. The first is a reserved byte indicating that the next two bytes have to be treated as a run length encoding. The second byte contains the run length character and the third byte contains the length of the run. Since we use a 3 byte compressed representation, only runs greater than 3 bytes are run length encoded.

The compression engine gets tokens from the parser. If the token is not a reserved code, it searches the hash tables to determine if it is a common word, uncommon word or literal and gets the equivalent numerical representation. This numerical representation is then written to the appropriate file and the inverted index is updated to indicate this occurrence of the keyword. This proceeds till the input text database has been scanned and compressed.

An added feature of the compression engine is its ability to maintain document boundaries in a multi document database. This allows us to reconstruct the original documents from a multi-document compressed database.

3.2 Search Engine

The basic operation of the search engine is quite simple. When the user enters a keyword for

searching we execute a hash based search function on it. If the keyword is found, the hash table returns the classification (common word, uncommon word or literal) and its numerical representation. Once we have the numerical code, we can index into the inverted index to get the linked list of pointers to occurrences of the keyword.

Since the inverted index is usually quite large, it is maintained on disk. When a keyword is found in the hash table, the corresponding inverted index entry is retrieved from disk. We have a block marker file associated with each compressed database, which marks the positions of the file pointers to the common word uncommon word and literal files at the start of each block.

To locate the first instance of a keyword, we go through the linked list to obtain the pointer to the block containing the keyword. Based on the information from the block marker file, we reposition the file pointers to the start of a block and perform a linear search to locate the keyword.

Typical queries ask for the first occurrence of a keyword, the next occurrence and so on based on the results obtained. A naive way to implement this would be to continue searching within the block till the occurrence number required by the user is found. For e.g. if user requests the third occurrence of *aardvark*, we search the block until we hit the third occurrence, ignoring the first two occurrences. While this solution works, it is hardly optimal.

In order to handle *next occurrence* kind of queries, we need maintain positional metadata on the previous location where the keyword was found. If we need to find the next occurrence of the keyword, we use this to reposition our *start of search* location. This solution allows us to perform incremental searching within a block with minimal time overhead.

After an instance of the keyword is found, we know the locations within the common word, uncommon word and literal files. We then backtrack on the common word, uncommon word and literal files till we can decompress a block of text (200 words in our case) around the occurrence and return it to the user. The backtracking algorithm is complicated by the fact that tokens in the common word file cannot be interpreted in the reverse order since there may be run length encoded tokens. Another complication is that we need to look out for document boundaries during the backtrack phase. In our scheme, backtracking is performed by using a lookback buffer for the common word file. This ensures that the numerical codes are interpreted correctly. The lookback buffer is not needed for the uncommon word and literal word files since these codes can be interpreted correctly in both directions.

3.3 Search Server Architecture

In our current implementation of the search engine, we have incorporated a communication sub-system, which allows queries to the search engine from the World Wide Web using a Java front-end. Communication is done using sockets opened between the Java client and the search server. For each query from a client, the server forks a copy of itself to perform the actual search and return the results. The server itself does a “busy wait” waiting for connections. The advantage of this scheme is that we use the semantics of the fork call to transfer the cache to the child process without performing a data copy. Most current UNIX systems implement the copy-on-write protocol. This reduces the overhead of the fork call. While this scheme allows a search child to see the same cache as the server, we need a mechanism to update the cache on the server when the child finds a new occurrence of a keyword. This is done by opening a named pipe between the server and its children. The children send back cache updates on the pipe and the server integrates it into its cache where it can be seen by future children. The update operation is sent as a block. Since the block is smaller than 4KB, UNIX named pipe semantics guarantee the atomicity of the update. This ensures that multiple simultaneous cache updates do not confuse the server.

As mentioned earlier in this section, we need to maintain file position metadata after each keyword search to optimize *next occurrence* queries. . We use a novel scheme to maintain this metadata. In the *SASE* server architecture, the Java clients who send in this information with each next occurrence query maintain this metadata. The major implication of this scheme is that the server is now stateless, since each client knows its version of the server state.

In our Web communication model, the Java clients open a TCP socket to the server and send in their query and the server metadata. To prevent the clients from using up all the server connections, the socket is closed once the search engine returns its results. The small lifetime of sockets combined with the stateless nature of the server allows us to shut down

Database	Original Size	*SASE* Size	Gzip Size	Comp. Ratio (*SASE*)	Comp. Ratio (Gzip)
Stories	6,944,363	3,125,644	2,625,350	54.99%	62.19%
RFC	68,940,062	28,610,430	18,272,426	58.49%	73.48%
News	314,572,800	166,411,289	110,463,639	47.09%	64.88%

Table 1: Comparison of compression ratios under *SASE* and GZIP. All file sizes are in bytes.

Database	Original Size	*SASE* Size	Glimpse Size	Comp. Ratio (*SASE*)	Comp. Ratio (Glimpse)
Stories	6,944,363	3,125,644	5,340,213	54.99%	23.10%
RFC	68,940,062	28,610,430	48,365,620	58.49%	29.84%

Table 2: Comparison of compression ratios under *SASE* and *Glimpse*. All file sizes are in bytes.

Database	Total Time	Linear search (iii)	Repositioning (iv)	Decompression (v)
Stories	27.8 ms	26.1 ms	220 us	1.4 ms
RFC	39.1 ms	36.8 ms	280 us	1.8ms
News	41.2 ms	38.8 ms	240us	1.7ms

Table 3: Timing breakdown of the various steps involved in a search. Block size is 8KB

the server and bring it up again without any noticeable difference to the clients. This is an attractive feature for administrative purposes. The downside of this scheme is that if the search database is updated, then the metadata on all the clients has to be updated as well. Since the clients close the socket once they get a response, we have no way of intimating them of changes in the server since we don't know who the clients are. This is solved by in a novel way. The compression engine generates a set of checksums when it compresses a database. These checksums are sent to the clients when they first query the server and the clients then send in this checksum to the server with each message. If the database on the server changes in between queries from a client, the checksum at the server would change as well. On the next query from a client, the checksum maintained by the client would not match the checksum at the server. The server then sends a flush message to the client, forcing the client to flush its old metadata. With this scheme built into a stateless server, *SASE* can handle very large inter query times without any server overhead.

4. Performance Results

In this section we report the results of a performance evaluation of the *SASE* prototype and an analysis of the results. To evaluate our compression efficiency, we compare *SASE* against GZIP one of the best lossless compression utilities. We also present the search performance numbers of *SASE*.

To ensure a representative text database, we chose [1]three large text databases each representing of a certain vocabulary. The first database consists of 7 MB of stories from Project Gutenberg. This represents everyday literary English usage. The second database contains the Internet RFC documents. This 70MB database represents technical vocabulary from a particular field, in this case networking. The third database consists of 300MB of USENET news articles from several different newsgroups from different newsdomains like alt, rec, misc and comp. This database contains vocabulary from a wide variety of domains.

4.1 Compression Performance

Table 1 compares the compression ratios of *SASE* and GZIP. Compression ratio is defined as:

$$\text{Compression Ratio} = 1 - \frac{\text{Compressed File Size}}{\text{Original File Size}}$$

[1] Availability of large text databases poses a problem in itself

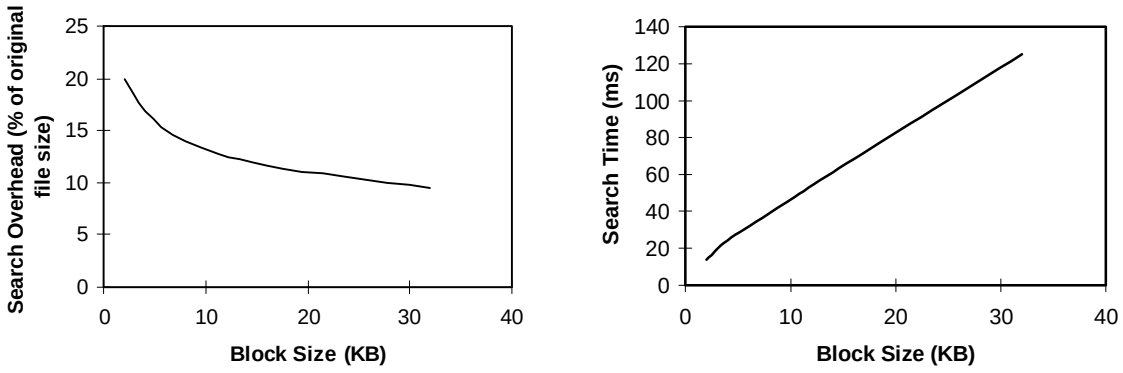

Figure 3: Effect of varying block size on the search overhead and query response time for the RFC database. Block size is varied betwen 2KB and 32KB.

The differences range between 7 and 17%. Although both *SASE* and GZIP are based on dictionary coding, GZIP can choose arbitrary length strings as candidate tokens for compression. Since *SASE* is limited to choosing strings demarcated by white spaces as tokens it suffers a performance penalty in compression. This performance gain in GZIP is more marked in the News and RFC databases, which have a more dynamic vocabulary.

4.2 Search performance

Execution of a search query under *SASE* involves (i) finding the numerical representation of the query keyword (ii) locating the first block containing the keyword (iii) performing a linear search to get the exact location of the keyword (iv) reposition of file pointers to begin decompression and (v) decompressing and transmitting the result back to the client. Steps (i) and (ii) are trivial and take much less time compared to the linear search required to locate the keyword within a block. Table 3 shows the timing breakdown of the various steps involved in a search.

As we mentioned in Section 2.2, by varying the block size, *SASE* allows a trade off between search time and index space overhead Figure 3 shows the effect of varying the block size on the RFC database. The index space overhead consists of the space taken up by the inverted indices for the uncommon word and literal dictionaries. The query search times were measured by searching for 10,000 random keywords and the runs were repeated for block sizes between 2KB and 32KB. Search times vary between 13ms to 120ms which compares favorably with a fully inverted index scheme.

5. Related Work

In [MOFF95][WITT94] [ZOBE95], Moffat and Zobel describe a word based lossless compression scheme which uses a fully inverted search index. The database is divided into files and compressed using Huffmann coding. Given a keyword, the inverted index is searched to get the linked list of occurrences. The portions of the file containing the keyword are then decompressed and sent to the user. Since this scheme uses a fully inverted index, the space taken up by the inverted index is much larger than *SASE*. Decompression speed is also slower than *SASE* due to the bit level manipulation that is required for decompressing Huffmann coded files. Variants of this scheme partition the database into blocks and compress the blocks using gzip. These schemes maintain inverted index pointers to blocks rather than every occurrence of a keyword. While these schemes have very good compression ratios, they need to decompress a block before searching through it. This operation increases their search time to several orders of magnitude greater than *SASE*.

The two step indexed approach taken by *SASE* is very similar to *Glimpse* [MANB94a][MANB94b]. Table 2 compares the compression efficiency of *SASE* and *Glimpse* (version 4.0) on both natural language text and technical documents. The difference in compression efficiency ranges between 28-31%. At the setting for the fastest search performance, the glimpse inverted index is 10%

larger than SASE for comparable search times. *SASE* also optimizes the two level approach with an index cache of dynamic block size which allows to use a large block size for space savings while retaining the speed advantages of a smaller block size. On the other hand, glimpse does better job in the choice of keywords to index. The approximate pattern matching algorithm in *glimpse* (*agrep*) is also more powerful than the simple keyword search mechanism of *SASE*. Since our VP tree based, approximate keyword match framework takes the string comparison function as a black box; *agrep* could be used to search the inverted index to provide similar pattern patching capability in *SASE*.

There are also several theoretical studies [AMIR96][FARA95] which discuss algorithms for searching through Lev-Zempel files. However, these schemes have not been implemented for us to make a fair comparison.

6. Conclusions

In this paper, we described a text search engine called *SASE*, which operates in the compressed domain. It provides an exact search mechanism using an inverted index and an approximate search mechanism using a vantage point tree. Secondly it allows a flexible trade-off between search time and storage space required to maintain the search indices. The results of our experiments show that the compression efficiency is within 7-17% of GZIP, which is one of the best lossless compression utilities. The sum of the compressed file size and the inverted indices is only between 55-76% of the original database, while the search performance is comparable to a fully inverted index.

We are currently working on the implementation of the approximate search mechanism using a vantage point tree. Another area of work is to use *SASE* as the underlying file system for NNTP servers. This gives NNTP servers the capability to perform keyword searches through USENET archives. When this system is up, it would yield important results on the choice of a cache replacement policy for the *SASE* dynamic index cache. While incremental additions to the compressed database are permitted in an inverted index based search system, the database is assumed to be mainly read-only. We are working on an indexing mechanism that would effectively remove the read-only restriction and allow the user to make real time changes to the database without having to recalculate the inverted indices. This scheme can be used for document management servers within companies and for maintaining the web page index database in Internet search engines like Lycos and AltaVista.

References

[**AMIR96**] Amir, A., Benson, G., Farach, M.; "Let sleeping files lie: pattern matching in Z-compressed files", *Journal of Computer and System Sciences (April 1996) vol.52, no.2, p. 299-307.*

[**BLUMER87**] Blumer A., Blumer J.; "On-Line Construction of a Complete Inverted File", *Technical Report, Dept. of Mathematics and Computer Science., University of Denver, CO*

[**BLUMER84**] Blumer A., Blumer J., Ehrenfeuchter A., Haussler D., McConnell R.; "Building a Complete Inverted File for a Set of Text Files in Linear Time", *Proceedings of the Sixteenth Annual ACM Symposium on the Theory of Computing.*

[**CHIU94**] Chiueh T.; "Content-based image indexing"
Proceedings of VLDB '94 pp. 582-593, Santiago Chile, September 1994

[**EVEN78**] Even S., Rodeh M.; "Economical Encoding of Commas Between Strings", *Communications of the ACM 21:4, 315-317*

[**FARA95**] Farach, M., Thorup, M.; "String matching in Lempel-Ziv compressed strings", *Proceedings of Symposium of Theory of Computing, pp. 703-712. Las Vegas, Nevada, USA.*

[**FALO85**] Faloutsos, C.; "Acccess methods for text",
ACM Computing Surveys, 17(March 1985), pp. 49-74.

[**FRAENKEL83**] Fraenkel A.S., Mor M.; "Is Text Compression by Prefixes and Suffixes Practical", *The Computer Journal 26:4, pp. 336-344*

[**KNUTH85**] Knuth D.E.; "Dynamic Huffman Coding",
Journal of Algorithms 6, pp. 163-180.

[**KNUTH77**] Knuth D.E. Morris J.H., Pratt V.R.; "Fast Pattern Matching in Strings",
SIAM Journal on Computing 6:2, pp. 323-349.

[**LEMPEL77**] Jacob Ziv, Abraham Lempel; "A Universal Algorithm for Sequential Data Compression", *IEEE Transactions on Information Theory, Vol. IT-23, No.3, May 1977.*

[**MANB94a**] Manber, U.; "A text compression scheme that allows fast searching directly in the compressed file", *Combinatorial Pattern Matching. 5th Annual Symposium, CPM 94. Proceedings, pp. 113-24. Asilomar, CA, USA, 5-8 June 1994.*

[**MANB94b**] Manber, U., Sun Wu; "GLIMPSE: a tool to search through entire file systems", *Proceedings of the Winter 1994 USENIX Conference, p. 23-32. San Francisco, CA, USA, 17-21 Jan. 1994.*

[**MCINTYRE85**] McIntyre D.R., Pechura M.A.; "Data Compression using Static Huffman Code-Decode Tables", *Journal of the ACM 28:6,612-616.*

[**MOFF95**] Moffat, A., Zobel, J.; "Information retrieval systems for large document collections", *Text Retrieval Conference (TREC-3), pp. 85-93. Gaithersburg, MD, USA, 2-4 Nov. 1994.*

[**PIKE81**] Pike J.; "Text Compression using a 4-bit Coding Scheme", *The Computer Journal 24:4, 324-330.*

[**STORER87**] Storer J.A., Tsang S.K.; "Data Compression Experiments Using Static and Dynamic Dictionaries", *Technical Report CS-84-118, CS Dept., Brandeis University Waltham, MA*

[**WAGNER73**] Wagner R.A.; "Common Phrases and Minimum-Space Text Storage", *Communications of the ACM 16:3, 148-152.*

[**WITT94**] Witten, I., Moffat, A., Bell, T.; "Managing Gigabytes", *Van Nostrand Reinhold, 1994.*

[**YANNAKOUDAKIS82**] Yannakoudakis E.J., Goyal P., Huggil J.A.; "The Generation and Use of Text Fragments for Data Compression", *Information Processing and Management 18:1, pp. 15-21.*

[**YIAN92**] P. Yianilos; "Data structures and algorithms for nearest neighbor search in general metric spaces", *Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms, pp. 311-321, Orlando, Fla., 1992.*

[**ZOBE95**] Zobel, J., Moffat, A.; "Adding compression to a full-text retrieval system", *Software - Practice and Experience (Aug. 1995) vol.25, no.8, pp. 891-903.*