RC25505 (WAT1410-098) October 29, 2014
Computer Science

IBM Research Report

Experimental Performance Evaluation to Enhance Database Compression on Commercial Servers

Hangu Yeo, Vadim Sheinin, Petros Zerfos
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598
USA

Experimental Performance Evaluation to Enhance Database Compression on Commercial Servers

Hangu Yeo, Vadim Sheinin and Petros Zerfos

IBM T. J. Watson Research Center
1101 Kitchawan Rd.
Yorktown Heights, NY 10598, USA
{hangu,vadims,pzerfos}@us.ibm.com

Abstract: Data compression technique is a very useful technique which reduces the redundancy within the data so that the same amount of information can be stored or transmitted in fewer number of bits. Data compression is widely used in data management, and can be broadly classified into hardware and software compressions. Commercial database servers are now designed with dictionary based proprietary compression algorithms as a software compression unit or as a hardware compression unit to reduce storage resources required on the server. In this paper, various data compression algorithms are evaluated with different configuration parameters and various input data formats to investigate a possibility to improve compression performance of dictionary based compression algorithms implemented on commercial database servers. Huffman coding algorithm produces an optimal prefix code tree and converts fixed-length symbols into variable-length code words. We cascaded Huffman coding with the dictionary based compression algorithm and evaluated the performance of the proposed algorithm with different configuration parameters using fabricated synthetic data as well as real customer data sets such as On-Line Transaction Processing (OLTP) benchmark TPC-E, decision support benchmark TPC-H and a plain text file as input data sets. Experimental results show that the compression can yield better compression when dictionary based compression algorithm is coupled with entropy coding algorithm such as Hoffman coding algorithm, and test benchmarks are compressed more efficiently. For example, the proposed compression method compresses OLTP benchmark tables at least 10% more efficiently. The evaluation also shows promising results in database benchmark compression using a single generic Huffman tree customized for a specific benchmark and generated using a set of input symbols randomly sampled from the corresponding benchmark data tables.

1. Introduction

The data compression techniques have been developed and improved over more than forty years, and have been used for applications in the area of data procession such as data storage and data transmission. The reduced amount of data to be stored or transmitted can reduce storage cost and increase the communication channel capacity. Data compression algorithms can be implemented in hardware or in software. Software compression is cheaper and more flexible solution of the two, but hardware implementation is important when compressing data needs to be done on the fly in real time or when software compression can degrade the performance of main processor. Data compression task consists of compression and decompression algorithms. There are two main types of data compression: lossless data compression and lossy data compression. Lossless data compression can reconstruct original data exactly from compressed bit stream. Lossy data compression cannot be decoded to reconstruct original data but produces data that is is close enough to original data. The lossy data compression can produce smaller compressed file than lossless data compression with better compression rate, but it is not a good method of compression of critical data. Lossless data compression can be classified into two groups. The first group of algorithms are dictionary based adaptive compression algorithms such as LZW (Lempel-Ziv-Welch) that does not depend on any pre-knowledge of the symbol statistics, and the second group of algorithms map input data to bit strings and generate bit sequences. Huffman coding and arithmetic coding algorithms are two well known algorithms belong to the second group.

LZW is a very well-known lossless data compression algorithm proposed by Lempel, Jacob Ziv, and Terry Welch [1] to improve the performance of LZ78 which is an alternate approach of the original Lempel-Ziv (LZ) technique published in 1977. LZW is a data compression technique that takes advantage of the repetition of characters within a data file. LZW is a dictionary based compression algorithm which encodes data by referencing a dictionary. LZW can compress any type of input files, but it generally performs better on files with repeated substrings. Huffman coding is an entropy encoding algorithm for lossless data compression proposed by Huffman [2] in 1952. Huffman coding is a variable-length coding scheme, is based on the frequency of occurrence of data and uses lower number of bits to encode the data that occurs more frequently. The variable-length bit sequences are stored in Huffman table which may be constructed for each individual input file or a set of input files. In all cases, Huffman table as well as the encoded bit strings needs to be stored (or transmitted), and the decoder uses the stored table to retrieve the corresponding original symbol sequence. Perl and Mehta [3] cascaded LZW and Huffman coding to improve text data compression. The combined compression algorithm is applied in the field of image processing to enhance coding efficiency of image file compression in [4] and [5]. The image data is compressed with Huffman coding in the first stage, and the compressed variable-length bit stream is compressed further with LZW compression algorithm in the second stage.

As discussed, the LZW algorithm is designed so that the same dictionary can be generated by the decoder and works well with repeated substring, and the Huffman coding is very efficient when the statistics of the input symbols to be coded are known in advance. The focus of this paper is to evaluate the two widely used lossless data compression algorithms and to propose an optimized algorithm which can take advantage of both LZW and Huffman coding. The proposed algorithm compresses input data with LZW in the first stage, and then Huffman coding is applied on the LZW fixed length output symbols. Fabricated synthetic table sets of TPC-E and TPC-H benchmarks [6] in flat file format and a plain text file are used as test input data sets. The main purpose of using TPC-E and TPC-H benchmark tables as input data sets is to test the proposed compression algorithm using a data set that would more accurately represent real world applications and OLTP systems. Real world customer data sets are also used to evaluate the proposed compressed algorithm as well. This paper is organized in four sections. Section 2 presents explanation of existing lossless data compression techniques. Two data compression schemes are considered, fixed-length and variable-length codeword compression schemes such as LZW and Huffman coding algorithms. The compression efficiencies of the algorithms are tested and compared using different configuration parameters and different types of input files. Section 3 presents two Huffman coding optimization methods which can enhance execution speed and storage requirement without significantly affecting compression ratio. For example, using a single customized generic Huffman table designed to compress benchmark table sets can reduce the disk space needed to store Huffman tables, and using reduced number of sampled symbols to build a Huffman tree can speed up the entire compression process. Section 3 also brings experiment results and performance analysis when the proposed optimized Huffman coding is applied to real customer data already compressed with a variant of LZW based data compression engine, and Section 4 concludes paper.

2. Evaluation of Lossless Data Compression Techniques

Abraham Lempel and Jacob Ziv published a paper on sliding window compression (LZ77) in 1977 and dictionary based compression technique (LZ78) in 1978. In 1984, Terry Welch improved LZ78, and developed a dictionary based compression technique (LZW) which still uses the same ideas of LZ78 and adds technical improvements over LZ78. The LZW adaptively changes dictionary of the strings that have appeared in the input file, and builds a dictionary that maps symbol sequences to fixed N-bit index of the dictionary. With the N bit index, the dictionary contains 2^N number of entries and the strings in the dictionary can have any length. For example, with 12-bit index, the dictionary may have 4,096 entries. The dictionary is dynamically created and the text strings are encoded by a reference to the dictionary. The dictionary is initialized with an entry for every possible byte with 256 entries, and other strings are built and added to the dictionary as the characters are read from an input file one after another. When the dictionary is fully populated, LZW no longer adds entries to the dictionary and compresses input characters using an unchanging dictionary. The first 256 entries of the dictionary are initialized to store all possible one byte data, and hence it guarantees that the input data is coded into a series of dictionary indexes. Typically the index length varies from 12-bit to 16-bit long. The dictionary does not need to be stored or transmitted because the dictionary can be reconstructed by the decoder while the compressed bit stream is being decompressed. When LZW is being implemented, the dictionary size needs to be decided carefully because the size of dictionary can affect the coding efficiency. For example, if the dictionary size is too big, coding efficiency may decrease because the code word size to encode also increases and every input character is encoded with longer code words. A common choice for the size of the dictionary is 4,096 entries with 12-bit index size. Table I compares coding efficiency of LZW with two different dictionary sizes, 12-bit dictionary and 16-bit dictionary. The number of bytes to fully populate 12-bit dictionary is measured as listed in the table. Less than 30 KB of input data is used to populate 4,096 entries of 12-bit dictionary. Three database tables (cash transaction, trade and trade history tables) of TPC-E benchmark and plain English text files are used as input data. Table II shows the compression results of same experiments (12-bit and 16-bit LZW compression) using TPC-H benchmark tables as input data.

Table I: Comparison of 12-bit and 16-bit LZW compression.

		TPC-E			English Text
		CashTran	Trade	TradeHistory	
Original File Size (Byte)		171,068,151	202,326,675	186,665,040	134,217,728
Bytes populate dictionary		15,942	13,603	28,609	10,418
12-bit LZW	Byte	61,828,998	88,029,010	74,027,881	62,755,287
	Save %	64	57	60	53
16-bit LZW	Byte	47,185,370	72,988,130	73,588,142	45,334,412
	Save %	73	64	61	66

Table II: Comparison of 12-bit and 16-bit LZW compression using TPC-H benchmark.

		TPC-H Benchmark			
		Customer	Fulfillment	Sales	Item
Original File Size (Byte)		15,728,640	326,344,704	255,062,016	666,132,480
12-bit LZW	Byte	8,162,832	113,295,537	121,498,356	257,510,531
	Save %	48	65	52	61
16-bit LZW	Byte	6,774,852	82,882,678	89,555,440	202,940,040
	Save %	57	74	65	69

Huffman coding is an entropy encoding algorithm for lossless data compression that uses less number of bits to encode the input symbol that occurs more frequently. Huffman coding uses variable-length coding technique and good estimation of probability of symbols in input is closely related to better performance in compression. Huffman tree is a binary tree which is built using the exact frequencies of the data and is built from bottom up instead of top down. The tree can be built using actual input data or using data which is representative of the entire input symbols. Huffman table can also be constructed by assigning a path from the root of the tree to a leaf node which corresponds to an input symbol in Huffman tree. Hence, each symbol is assigned a variable-length bit string, and a bit string in Huffman table cannot be the prefix of another bit string. The Huffman lookup table is used to encode or decode the Huffman bit stream. Unlike LZW compression algorithm, Huffman tree (or Huffman table) needs to be stored or transmitted to the receiver, and the extra cost of storing or transmitting the Huffman tree may degrade the performance of the compression. It may be possible to choose a representation of the binary Huffman tree and decode any bit stream by traversing the tree. Furthermore, building a Huffman tree using all input symbols can slow down whole compression process, and can be improved by populating the table with subset of input symbols. We implemented Huffman coding in conjunction with 12-bit and 16-bit LZW compression algorithm so that Huffman coding unit takes the output symbols of LZW algorithm and generates compressed variable-length bit streams. Table III and Table IV compares coding efficiency of 12-bit and 16-bit LZW and Huffman combined compression algorithm. Table V shows that LZW and Huffman combined algorithm compresses up to 35% further than LZW only compression.

Table III: Comparison of 12-bit and 16-bit compression using concatenated LZW and Huffman compression algorithms.

		TPC-E			English Text
		CashTran	Trade	TradeHistory	
Original File Size (Byte)		171,068,151	202,326,675	186,665,040	134,217,728
12-bit LZW + Huffman	Byte	51,425,359	75,810,350	53,243,482	57,234,912
	Save %	70	62	71	57
16-bit LZW + Huffman	Byte	37,465,323	60,547,986	47,867,220	39,650,327
	Save %	78	70	74	70

Table IV: Comparison of 12-bit and 16-bit compression using concatenated LZW and Huffman compression algorithms using TPC-H benchmark.

		TPC-H Benchmark			
		Customer	Fulfillment	Sales	Item
Original File Size (Byte)		15,728,640	326,344,704	255,062,016	666,132,480
12-bit LZW + Huffman	Byte	6,997,421	99,310,184	104,795,637	225,990,015
	Save %	69	62	59	66
16-bit LZW + Huffman	Byte	5,996,036	76,096,034	81,082,546	189,855,619
	Save %	62	77	68	71

Table V: Comparison of LZW only compression (Table I and Table II) and LZW and Huffman combined compression (Table III and Table IV).

	TPC-E Benchmark			English Text
	CashTran	Trade	TradeHistory	
12-bit LZW+Huffman Save (%)	17	14	28	9
16-bit LZW+Huffman Save (%)	21	18	35	13
	TPC-H Benchmark			
	Customer	Fulfillment	Sales	Item
12-bit LZW+Huffman Save (%)	14	12	14	12
16-bit LZW+Huffman Save (%)	12	8	10	7

Huffman coding produces best results when the probabilities of symbols' occurrence are powers of ½. Otherwise, more bits are spent on encoding a symbol than theoretical entropy bound. Huffman coding is inefficient especially when the probability of a symbol is large because it only assigns integer number of bits for each symbol to be encoded. For example, if the probability of a symbol is 0.9, the codeword length should be 0.152 bit in theory, but Huffman coding only assigns one bit to the symbol. Block Huffman coding [7] was originally developed to enhance the coding efficiency when the probability is skewed. The Block Huffman Coding groups multiple symbols into a new extended symbol (N) and Huffman tree is built based on the probability of block of symbols instead of using probability of individual symbols. This method splits large probability value corresponding to a single symbol into many symbols with smaller values of probability, and hence allows better entropy closer to the Shannon entropy bound [8]. For example, 12-bit input symbols (4,096 total number of symbols) in the above example are grouped into 2-symbol blocks (N = 2), and Huffman tree is built based on the probability of 2-symbol blocks (4,096 x 4,096) as depicted in Figure 1. Table VI shows that the compression ratio improves when regular Huffman coding (N = 1) is replaced with Block Huffman coding with N=2, but the compression and decompression complexity increases exponentially from 2^{12} to 2^{24}, which makes the block Huffman coding somewhat too complicated to implement. The Huffman tree size and Huffman code length increase exponentially as the value N increases as well.

$$\{ S_0, S_1, S_2, S_3, \ldots, S_{4095} \}$$

(a)

$$\begin{aligned} \{ & S_{0,0}, S_{0,1}, S_{0,2}, S_{0,3}, \ldots, S_{0,4095}, \\ & S_{1,0}, S_{1,1}, S_{1,2}, S_{1,3}, \ldots, S_{1,4095}, \\ & \ldots\ldots\ldots\ldots \\ & S_{4095,0}, S_{4095,1}, S_{4095,2}, S_{4095,3}, \ldots, S_{4095,4095} \} \end{aligned}$$

(b)

Figure 1: (a) Original Huffman symbols. (b) Block Huffman Symbols with N = 2.

Table VI: Comparison of Regular Huffman coding and Block Huffman coding in conjunction with 12-bit LZW compression.

	TPC-E			English Text
	CashTran	Trade	TradeHistory	
12-bit LZW + Huffman	51,425,359	75,810,350	53,243,482	134,217,728
12-bit LZW + Block Huffman with N=2	36,767,921	56,345,397	42,330,188	57,234,912
Save (%)	29	26	20	57

3. Huffman Coding Enhancement

Huffman coding produces lossless compressed bit stream and is relatively easy to implement. However, compression efficiency of the algorithm mainly depends on accuracy of the frequencies and probabilities of the symbols. Therefore, in order to achieve an efficient compression rate, Huffman tree needs be built in advance using all symbols to be encoded in the input file before the symbols are actually encoded using the tree. Since compression symbol frequencies vary with input context, Huffman tree needs to be rebuilt individually for each input file when each input file is being compressed. Furthermore, it is required to store or transmit Huffman tree corresponding to a compressed file with compressed bit stream so that the decoder uses the tree to decompress the bit stream. However, generating a customized Huffman tree for each table may require too much memory space to store all generated trees for a database benchmark. Instead of customizing a Huffman tree that works best to compress an individual table of a benchmark, we investigate whether a generic Huffman tree may work for all tables of a benchmark. This can relieve the burden of generating and storing customized Huffman tree corresponding to each individual table. Table VII shows that LZW and Huffman with generic Huffman tree combined algorithm still generates compressed benchmark tables up to 30% smaller than compressed benchmark tables compressed with LZW only. However, the compression improvement is less than customized Huffman table as shown in Table V.

Table VII: Comparison of LZW only compression (Table I) and LZW and Huffman combined compression algorithm. One generic Huffman tree is used to compress all three tables.

	TPC-E		
	CashTran	Trade	TradeHistory
12-bit LZW + Generic Huffman	56,489,510	80,590,485	59,392,494
12-bit Generic Huffman Save (%)	8.7	8.5	19.8
16-bit LZW + Generic Huffman	41,750,758	64,561,120	52,370,048
16-bit Generic Huffman Save (%)	11.5	11.5	28.8

Generating a Huffman tree sometimes takes too much time and can be a time consuming process especially when the number of symbols generated by the preceding LZW algorithm is too many. Instead of waiting for the LZW algorithm to produce all encoded symbols, and using all input symbols to build a Huffman tree, Huffman tree can be built using only representative of symbols that the preceding LZW algorithm outputs. We investigate a method of building a Huffman tree using subset of output symbols to reduce complexity of building Huffman tree without sacrificing coding efficiency. We evaluated the performance of Huffman trees generated using different amount of random and non-random samples of output symbols, and the comparison of the evaluation results are listed in Table VIII. The percentage of sample varies from 1% to 80%. Table VIII compares the compressed table sizes, and shows that randomly sampled symbols are better representatives of the entire input data than non-randomly selected symbols which are selected from the beginning of the input symbols in order.

Table VIII: Comparison of building Huffman tree using different number of sampled symbols.

	Non-Randomly Sampled Symbols			Randomly Sampled Symbols		
	Cash Tran	Trade	Trade Hist	Cash Tran	Trade	Trade Hist
1 %	57,126,585	80,762,868	71,185,437	49,750,965	74,094,318	47,478,578
2 %	57,983,383	80,987,288	73,921,508	49,704,268	74,060,790	47,450,144
5 %	55,093,961	79,789,089	66,259,532	49,685,148	74,047,710	47,432,147
10 %	54,388,785	79,429,938	64,275,872	49,680,479	74,043,189	47,427,361
20 %	53,303,189	77,829,442	60,859,736	49,678,741	74,040,418	47,424,918
50 %	53,309,758	77,660,242	61,332,166	49,677,630	74,039,271	47,423,973
60 %	52,614,396	77,060,290	59,308,978	49,677,420	74,039,166	47,423,763
70 %	51,905,256	76,436,039	55,654,418	49,677,409	74,038,969	47,423,785
80 %	51,536,193	75,991,127	53,830,255	49,677,469	74,038,974	47,423,513
100 %	51,425,359	75,810,350	53,243,482	49,677,221	74,038,864	47,423,370

Data compression algorithms implemented on database servers are mainly based on a variant of dictionary based LZW data compression to compress large amount of relational data. The dictionaries are populated using a small number of data records. When the dictionaries are fully populated, the algorithm becomes non-adaptive, and the dictionaries are unchanged while the compression lasts. To enhance compression efficiency of dictionary based data compression, we cascaded the dictionary based compression with Huffman coding, evaluate the compression performance and compared the evaluated results with the dictionary only based compression technique. Real world customer tables are compressed with database compression engine configured with various dictionary sizes (symbol lengths) to produce 11-bit, 12-bit and 13-bit compressed symbols. The symbols are indexes of the appropriate entry in the dictionary with 2K, 4K and 8K entries. The compressed symbols are extracted, and only 1% of the extracted symbols are randomly selected and used to build Huffman tree. The extracted symbols are further compressed into variable-length bit strings using Huffman tree built with randomly sampled symbols. The evaluation procedure is depicted in Figure 2. Since not all the entries of the dictionary have the same frequency, Huffman coder assigns variable number of bits to each symbol. Table IX compares compressed file sizes using customer files and shows that additional Huffman coding enhances compression up to 24%.

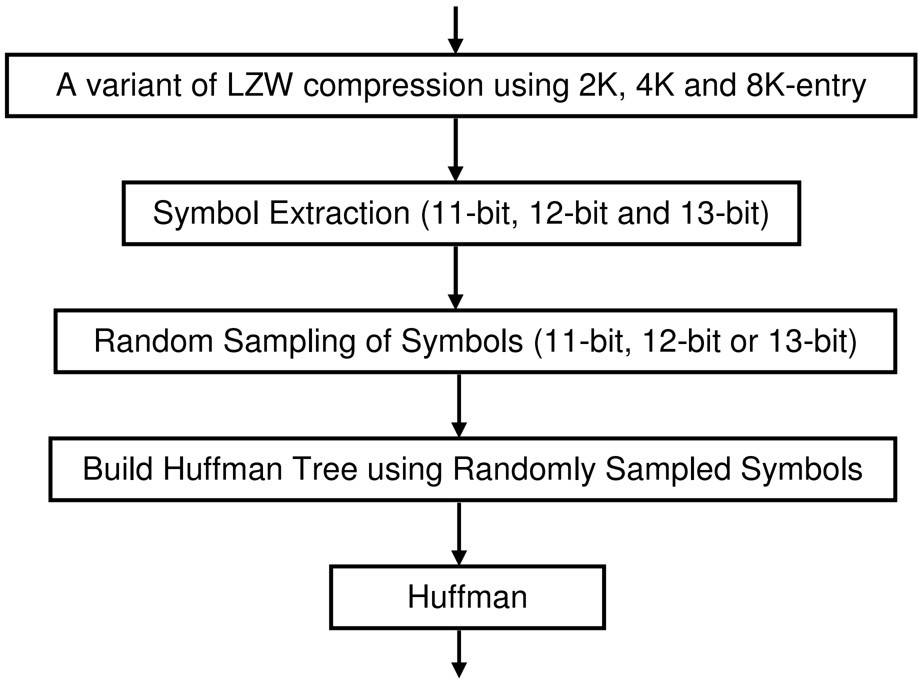
A variant of LZW compression using 2K, 4K and 8K-entry
Symbol Extraction (11-bit, 12-bit and 13-bit)
Random Sampling of Symbols (11-bit, 12-bit or 13-bit)
Build Huffman Tree using Randomly Sampled Symbols
Huffman

Figure 2: Evaluation process of proposed compression scheme using customer data compressed with database compression engine.

Table IX: Comparison of dictionary based only compression and dictionary based and Huffman coding cascaded compression.

	Customer File 1	Customer File 2	Customer File 3
Dictionary based Compression	107,560,960	97,091,584	68,116,480
Symbol Size	12 bit	12 bit	12 bit
With cascaded Huffman coding	80,916,480	81,207,296	55,517,184
Huffman Save (%)	24 %	16 %	18 %

	Customer File 4	Customer File 5	Customer File 6
Dictionary based Compression	710,455,638	43,735,693	189,567,735
Symbol Size	13 bit	11 bit	11 bit
With cascaded Huffman coding	627,775,932	37,835,685	146,365,047
Huffman Save (%)	12 %	13 %	23 %

4. Conclusions

Data compression is the most effective way to save on storage requirement transmission time and bandwidth usage in large scale database applications. Storage cost is one of the key costs in relational database systems despite the drop in prices, and data compression technique can maximize the use of disk space and reduce the storage requirement. Therefore, commercial database servers usually include their own data compression and decompression engines to efficiently compress and decompress the database tables. In this paper, we evaluated two widely used source coding algorithms, LZW and Huffman coding algorithms, with fabricated synthetic benchmark tables (TPC-E and TPC-H benchmarks) and real world customer data. The test results show that 16-bit LZW compression shows better compression than 12-bit LZW with every test benchmark. We also proposed and evaluated an entropy coding based Huffman coding algorithm concatenated with the existing a variant of LZW dictionary based compression scheme. To reduce the additional storage requirement and execution time that can be possibly introduced by adding the additional compression unit, we proposed a customized generic Huffman table built with a small set of input symbols (about 1% of input symbols are sampled) randomly sampled from compressed TPC-E and TPC-H benchmark tables respectively, and used the generic table to compress all tables of each benchmark without loosing compression efficiency much. Our preliminary results with real customer data suggest that the compressed file size can be further reduced by at least 10 % although Huffman tree overhead (about 5%) and the fact that Huffman coding performance is closely related with input symbol statistic accuracy are taken into account. We believe the 10% of data reduction can be significant when large databases are considered, and can contribute to the reduction in database server operation cost.

References

[1] T. Welch, "A Technique for High Performance Data Compression," Computer 17 (6), pp. 8-19, 1984.

[2] D. A. Huffman, "A Method for the Construction of Minimum-Redundancy Codes," Proceedings of the I.R.E., pp. 1098-1101, September 1952.

[3] Y. Perl and A. Mehta, "Cascading LZW Algorithm with Huffman Coding: A Variable to Variable Length Compression Algorithm," Proceedings of the First Great Lakes Computer Science Conference on Computing in the 90's, pp. 170-178, 1989.

[4] C. Saravanan and M. Surender, "Enhancing Efficiency of Huffman Coding using Lempel Ziv Coding for Image Compression," International Journal of Soft Computing and Engineering (IJSCE), vol. 2, pp. 38-41, January 2013.

[5] D. Kaur and K. Kaur, "Huffman Based LZW Lossless Image Compression Using Retinex Algorithm," International Journal of Advanced Research in Computer and Communication Engineering, vol. 2, pp. 3145-3151, August 2013.

[6] TPC Transaction Processing Performance Council, http://www.tpc.org/.

[7] M. A. Mannan and M. Kaykobad, "Block Huffman Coding," Computers and Mathematics with Applications, vol. 46, pp. 1581-1587, November 2003.

[8] C. E. Shannon, "A Mathematical Theory of Communication," Bell System Technical Journal, Vol. 27, pp. 379-423, July 1948.