

A Compressed Index-Query Web Search Engine Model

Hussein Al-Bahadili
Faculty of Information Technology
Petra University
Amman, Jordan

Saif Al-Saab
Faculty of Information Technology & Systems
The Arab Academy for Banking & Financial Sciences
Amman, Jordan

Abstract— In this paper, we propose a new web search engine model based on index-query bit-level compression. The model incorporates two bit-level compression layers both implemented at the back-end processor (server) side, one layer resides after the indexer acting as a second compression layer to generate a double compressed index, and the second layer be located after the query parser for query compression to enable bit-level compressed index-query search. This contributes to reducing the size of the index file as well as reducing disk I/O overheads, and consequently yielding higher retrieval rate and performance. The data compression scheme used in this model is the adaptive character wordlength (ACW(n,s)) scheme, which is an asymmetric, lossless, bit-level scheme that permits compressed index-query search. Results investigating the performance of the ACW(n,s) scheme is presented and discussed.

Keywords- Web search engine model; index files; query optimization; full-text compressed self-index; bit-level compression; ACW(n,s) scheme.

I. INTRODUCTION

A Web search engine is an information retrieval system designed to help finding information stored on the World Wide Web (or simply the Web) [1]. The Web search engine allows us to search the Web storage media for a certain content in a form of text meeting specific criteria (typically those containing a given word or phrase) and retrieving a list of files that match those criteria [2]. Standard Web search engine consists of three main components: Web crawler, document analyzer and indexer, and search processor [3].

Due to the rapid growth in the size of the Web, Web search engines are facing enormous performance challenges, in terms of: storage capacity, data retrieval rate, query processing time, and communication overhead. Large search engines, in particular, have to be able to process tens of thousands of queries per second on tens of billions of documents, making query throughput a critical issue [4]. To satisfy this heavy workload, search engines use a variety of performance optimizations including succinct data structure [5, 6], compressed text indexing [7, 8], query optimization [9, 10], high-speed processing and communication systems [11], and efficient search engine architectural design [12].

Compressed text indexing has become a popular alternative to cope with the problem of giving indexed access to large text collections without using up too much space.

Reducing space is important because it gives the chance of maintaining the whole collection of data in main memory. The current trend in compressed indexing is full-text compressed self-indexes [13]. Such a self-index replaces the text by providing fast access to arbitrary text substrings, and, in addition, gives indexed access to the text by supporting fast search for the occurrences of arbitrary patterns. However, we believe that the performance of the current search engine models based on compressed text indexing techniques still short from meeting users and applications needs.

In this work we propose a new web search engine model that is based on index-query bit-level compression. The model incorporates two bit-level compression layers both implemented at the back-end processor side, one after the indexer acting as a second compression layer to generate a double compressed index, and the other one after the query parser for query compression to enable bit-level compressed index-query search. So that less disk space is required storing the index file, reducing disk I/O overheads, and consequently higher retrieval rate or performance.

One important feature of the bit-level technique to be used for performing the search process at the compressed index-query level is to generate similar compressed binary sequence for the same character from the search queries and the index files. A data compression scheme that satisfies this requirement is the recently developed adaptive character wordlength (ACW(n,s)) scheme [14, 15], therefore, it will be used in this work. Recent investigations on using this scheme for text compression demonstrated an excellent performance in comparison with many widely-used data compression algorithms and state-of-the-art tools [14, 15].

This paper is organized in seven sections. Section I provides an introduction to the general domain of the paper. The rest of this paper is organized as follows: Section II presents some of the most recent and related work. The components of the standard web search engine model are described in Section III. A detail description of the lossless bit-level data compression scheme that shall be used in this model, namely, the ACW(n,s) scheme is given in Section IV. The proposed search engine model is described in Section V. Experimental results on using the ACW(n,s) scheme for text compression are presented and discussed in Section VI. Finally, in Section VII, based on the results obtained, conclusions and recommendations for future work are pointed-out.

II. LITERATURE REVIEW

In this section we present a review on the most recent work related to Web search engine. E. Moura et. al. [16] presented a technique to build an index based on suffix arrays for compressed texts. They proposed a compression scheme for textual databases based on generating a compression code that preserves the lexicographical ordering of the text words. As a consequence it permits the sorting of the compressed strings to generate the suffix array without decompressing. Their results demonstrated that as the compressed text is under 30% of the size of the original text, they were able to build the suffix array twice as fast on the compressed text. The compressed text plus index is 55-60% of the size of the original text. In addition, the technique reduced the index and search times to approximately half the time.

S. Varadarajan and T. Chiueh [17] described a text search engine called shrink and search engine (SASE), which operates in the compressed domain. SASE provides an exact search mechanism using an inverted index and an approximate search mechanism using a vantage point tree. The SASE allows a flexible trade-off between search time and storage space required to maintain the search indices. The experimental results showed that the compression efficiency is within 7-17% of GZIP. 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.

R. Grossi et. al. [18] presented an implementation of compressed suffix arrays exhibiting tradeoffs between search time and space occupancy for a given text (or sequence) of n symbols over an alphabet Σ , where each symbol is encoded by $\log |\Sigma|$ bits. They showed that compressed suffix arrays use just $nH_h + O(n \log \log n / \log_{|\Sigma|} n)$ bits, while retaining full text indexing functionalities, such as searching any pattern sequence of length m in $O(m \log |\Sigma| + \text{polylog}(n))$ time.

X. Long and T. Suel [10] studied pruning techniques that can improve query throughput and response times for query execution in large engines in the case where there is a global ranking of pages, as provided by Page rank or any other method, in addition to the standard term-based approach. Their results showed that there is significant potential benefit in such techniques.

V. N. Anh and A. Moffat [19] described a scheme for compressing lists of integers as sequences of fixed binary codewords that had the twin benefits of being both effective and efficient. Because Web search engines index large quantities of text, the static costs associated with storing the index can be traded against dynamic costs associated with using it during query evaluation. The approach resulted in a reduction in index storage costs compared to their previous word-aligned version, with no cost in terms of query throughput.

P. Ferragina et. al. [13] proposed two new compressed representations for general sequences, which produce an index that improves over the one in [Gro 03] by removing from the query times the dependence on the alphabet size and the poly logarithmic terms.

R. Gonzalez and G. Navarro [20] introduced a new compression scheme for suffix arrays which permits locating the occurrences extremely fast, while still being much smaller than classical indexes. In addition, their index permits a very efficient secondary memory implementation, where compression permits reducing the amount of I/O needed to answer queries. Later on in [21], they improved their work by introducing a practical disk-based compressed text index that, when the text is compressible, takes little more than the plain text size (and replaces it). It provides very good I/O times for searching, which in particular improve when the text is compressible. They showed experimentally that their index is extremely competitive on compressible texts.

A. Moffat and J. S. Culpepper [22] showed that a relatively simple combination of techniques allows fast calculation of Boolean conjunctions within a surprisingly small amount of data transferred. This approach exploits the observation that queries tend to contain common words, and that representing common words via a bitvector allows random access testing of candidates, and, if necessary, fast intersection operations prior to the list of candidates being developed. By using bitvectors for a very small number of terms that (in both documents and in queries) occur frequently, and byte coded inverted lists for the balance can reduce both querying time and also query time data-transfer volumes. The techniques described in [22] are not applicable to other powerful forms of querying. For example, index structures that support phrase and proximity queries have a much more complex structure, and are not amenable to storage (in their full form) using bitvectors.

P. Ferragina et. al. [8] presented an article to fill the gap between implementations and focused comparisons of compressed indexes. The existing implementations of compressed indexes offers tuned implementations and a standardized API for the most successful compressed full-text self-indexes, together with effective test-beds and scripts for their automatic validation and test; and they performed experiments on a number of codes with the aim of demonstrating the practical relevance of this technology.

III. SEARCH ENGINE MODEL

Standard Web search engine consists of the following main components [2, 3]: Web crawler, document analyzer and indexer, and searching process. In what follows, we briefly describe each of the above components.

A. Web Crawler

A Web crawler is a computer program that browses the Web in a methodical, automated manner. Other terms for Web crawlers are ants, automatic indexers, bots, and worms or Web spider, Web robot. Unfortunately, each spider has its own personal agenda as it indexes a site. Some utilize META tag keywords; another may use the beta description of a site, and some use the first sentence or paragraph on the sites homepage. In other words, a page that ranks well on one search engine may not rank as well on another. Given a set of "seed" URLs, the crawler repeatedly removes one URL from the seeds, downloads the corresponding page, extracts all the URLs contained in it, and adds any previously unknown URLs to the seeds [1].

Web search engines work by storing information about many Web pages, which they retrieve from the Web itself. These pages are retrieved by a spider - sophisticated Web browser which follows every link extracted or stored in its database. The contents of each page are then analyzed to determine how it should be indexed, for example, words are extracted from the titles, headings, or Meta tags.

B. Document Analyzer and Indexer

Indexing is the process of creating an index that is a specialized file containing a compiled version of documents retrieved by the spider [1]. Indexing process collect, parse, and store data to facilitate fast and accurate information retrieval. Index design incorporates interdisciplinary concepts from linguistics, mathematics, informatics, physics and computer science [12].

The purpose of storing an index is to optimize speed and performance in finding relevant documents for a search query. Without an index, the search engine would scan every (possible) document in the Internet, which would require considerable time and computing power (impossible with the current Internet size). For example, while an index of 10000 documents can be queried within milliseconds, a sequential scan of every word in the documents could take hours. The additional computer storage required to store the index, as well as the considerable increase in the time required for an update to take place, are traded off for the time saved during information retrieval [2].

1) Index design factors

Major factors in designing a search engine's architecture include the following [1-3]:

- *Merge factors*: How data enters the index, or how words or subject features are added to the index during text corpus traversal, and whether multiple indexers can work asynchronously. The indexer must first check whether it is updating old content or adding new content. Traversal typically correlates to the data collection policy. Search engine index merging is similar in concept to the sequential language (SQL) Merge command and other merge algorithms.
- *Storage techniques*: How to store the index data, i.e., whether information should be compressed or filtered.
- *Index size*: How much computer storage is required to support the index.
- *Lookup speed*: How quickly a word can be found in the index. The speed of finding an entry in a data structure, compared with how quickly it can be updated or removed, is a focus of computer science.
- *Maintenance*: How the index is maintained over time.
- *Fault tolerance*: How important it is for the service to be reliable. Issues include dealing with index corruption, determining whether bad data can be treated in isolation, dealing with bad hardware, partitioning, and schemes such as hash-based or composite partitioning, as well as replication.

2) Index data structures

Search engine architectures vary in the way indexing is performed and in methods of index storage to meet the various design factors. There are many architectural designs for the indexes and the most widely-used one is inverted index [11-12]. Inverted index stores a list of occurrences of each atomic search criterion, typically in the form of a hash table or binary tree.

The inverted index structure is widely use in the modern supper fast search engine like Google, Yahoo, Lucene and other major Web search engines. Through the indexing, there are several processes taken place, here the processes that related to our work will be discussed. These processes may be used and this depends on the search engine configuration [1-3, 23].

- *Extract URLs*. A process of extracting all URLs from the document being indexed, it used to guide crawling the website, do link checking, build a site map, and build a table of internal and external links from the page.
- *Code striping*. A process of removing HTML tags, scripts, and styles, and decoding HTML character references and entities used to embed special characters.
- *Language recognition*. A process by which a computer program attempts to automatically identify, or categorize, the language or languages of a document.
- *Document tokenization*. A process of detecting the encoding used for the page; determining the language of the content (some pages use multiple languages); finding word, sentence and paragraph boundaries; combining multiple adjacent-words into one phrase; and changing the case of text.
- *Document parsing or syntactic analysis*. The process of analyzing a sequence of tokens (for example, words) to determine their grammatical structure with respect to a given (more or less) formal grammar.
- *Lemmatization/stemming*. The process for reducing inflected (or sometimes derived) words to their stem, base or root form – generally a written word form, this stage can be done in indexing and/or searching stage. The stem need not be identical to the morphological root of the word; it is usually sufficient that related words map to the same stem, even if this stem is not in itself a valid root. The process is useful in search engines for query expansion or indexing and other natural language processing problems.
- *Normalization*. The process by which text is transformed in some way to make it consistent in a way which it might not have been before. Text normalization is often performed before text is processed in some way, such as generating synthesized speech, automated language translation, storage in a database, or comparison.

C. Searching Process

When the index is ready the searching can be performed through query interface, a user enters a query into a search engine (typically by using keywords), the engine examines its index and provides a listing of best matching Web pages according to its criteria, usually with a short summary containing the document's title and sometimes parts of the text. In this stage the results are ranked, where ranking is a relationship between a set of items such that, for any two items, the first is either "ranked higher than", "ranked lower than" or "ranked equal" to the second.

In mathematics, this is known as a weak order or total pre-order of objects. It is not necessarily a total order of documents because two different documents can have the same ranking. Ranking is done according to document relevancy to the query, freshness and popularity [3]. Fig. 1 outlines the architecture and main components of standard search engine model.

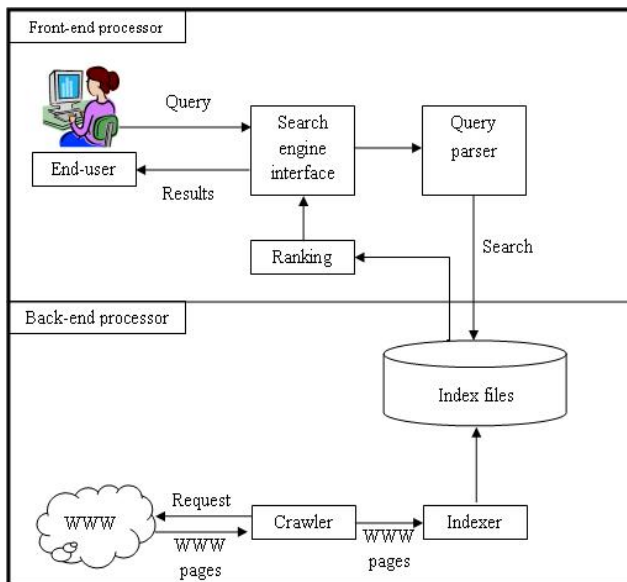


Figure 1. Architecture and main components of standard search engine model.

IV. THE DATA COMPRESSION SCHEME

This section presents a description of the data compression scheme that shall be used in this search engine model, namely, the adaptive character wordlength (ACW(n,s)) scheme [14]. This scheme is based on the adaptive character wordlength (ACW(n)) algorithm [15], therefore, it will be described first.

A. The ACW(n) Algorithm

In the ACW(n) algorithm [15], first, the source file is converted to binary sequence using certain character-to-binary coding format (e.g., the well-known ASCII coding, adaptive coding [14, 24], etc.). The binary sequence is then subdivided into blocks of n -bit length. The equivalent decimal value for each block is calculated and if the total different decimal values (d) is ≤ 256 , compression can be performed using n -bit character wordlength. Otherwise, another value of n should be examined.

This iterative process continues until a specific value of n satisfies the above condition, otherwise the binary sequence cannot be compressed. Therefore, this algorithm is referred to as ACW(n) algorithm.

If a specific value of n satisfies the above condition, the different decimal values are sorted in ascending or descending order. So that, each block is then converted to character according to its sequence number and not according to its actual decimal value. In order not to get the binary sequence mixed up during the decompression process, these sorted values should be stored with other information (e.g., n , d , etc.) at the compressed file header.

The probability of satisfying the condition of $d \leq 256$ is inversely proportional to n and it is given as $p = 2^{8-n}$. For example, the probabilities of $d \leq 256$, for $n=9$ and $n=10$, are 0.5 and 0.25, respectively. The main issues of the ACW(n) algorithm can be summarized as follows:

- The binary sequence can be compressed using n -bit character wordlength only if $d \leq 256$.
- The probability of being able to successfully compress the binary sequence using n -bit character wordlength is inversely proportional to n .
- Finding the optimum value for n that provides the maximum compression ratio is time consuming process, especially for large binary sequences.
- Converting text to binary using the ASCII code yields a high entropy binary sequence, which means either small or no compression can be achieved.

However, these issues can be considered as drawbacks that may degrade the performance of the ACW(n) algorithm.

B. The ACW(n,s) Scheme

To enhance the performance of the ACW(n) algorithm and prevail over all consequences of the above issues, the binary sequence is subdivided into a number of subsequences (s), each of them satisfies the condition that $d \leq 256$. Therefore, the new scheme is referred to as ACW(n,s). Subdividing the original binary sequence into subsequences eliminates the first downside of $d \leq 256$ and consequently the worry of having low success probability, if large value for n is used.

To further enhance the performance of the ACW(n,s) algorithm, an adaptive coding format was introduced in which the original characters are coded to binary according to their probability of occurrence. This coding format reduces the entropy of the binary sequence so that a higher compression ratio is granted. Definition of entropy and how does adaptive coding is working are given in [14].

C. The Compressed File Header

In order not to get the codes mixed up during the decompressing process, some information needs to be stored in the compressed file header as shown in Fig. 2. The size of the header is directly proportional to the number of subsequences; therefore, an optimization mechanism is required to find the values of n and s that yields a maximum compression ratio.

| | | | | | | | |
|------------------------------------|---|-------------------------------------|---|-------|-----|-------------------------------|--|
| ACW field S_{ACW} (8-byte) | Coding field S_F (Coding dependent) | Sequence field S_S (8-byte) | Subsequences fields $S_{Sub}=s(8+L)$ | | | | Characters coded according to their subsequence number using n-bit character wordlength C_1, C_2, \dots, C_B |
| | | | SSFH ₁ (8-byte) | L_1 | ... | SSFH _i (8-byte) | |

Figure 2. The compressed file using the ACW(n,s) scheme.

The compressed file header of the ACW(n,s) scheme consists of the following fields:

- 1) ACW field (8 Bytes)
- 2) Coding field (Coding format dependent)
- 3) Sequence field (8 Bytes)
- 4) Subsequence fields (ACW(n,s) dependent)

In what follows a description is given for each of the above fields.

1) ACW field (S_{ACW})

The ACW field (S_{ACW}) is an 8-byte field encloses information related to the ACW(n,s) algorithm, such as: algorithm name (ACW), algorithm version, coding format, character wordlength. The constituents of this field are kept similar to the first 8-bytes in the ACW(n) algorithm. Table 1 lists the constituents of this field and their description.

2) Coding field (S_F)

The coding field (S_F) encloses information related to the coding format, so that their content depends on the coding format indicated in the ACW field. This field is not required for the ASCII coding format. For the adaptive coding, it contains the number of characters within the source file and a list of these characters after sorting (N_c+1), while for Huffman coding; it enfolds the same components as in standard Huffman compression algorithm [24].

3) Sequence field (S_S)

The sequence field (S_S) is an 8-byte field that contains the number of subsequences in the first 4-bytes, and the last 4-bytes remain unused for future development.

4) Subsequence field (S_{Sub})

The subsequence fields (S_{Sub}) enfold the subsequences information. It consists of two subfields. The first one is a fixed 8-byte subfield that includes the subsequence number (4-byte) and the number of the different decimal values within the subsequence (d) (1-byte). Three bytes are remained unused for future development.

The second subfield has a variable length, which is referred to as L , and it contains the different decimals values of the blocks. The total length of the subsequence fields is given by:

$$S_{Sub} = s(8 + L) \quad (1)$$

$$L = \begin{cases} \frac{2^n}{8} & \text{for } n \leq 11 \\ d \left\lceil \frac{n}{8} \right\rceil & \text{for } n \geq 12 \end{cases} \quad (2)$$

As shown in Fig. 2, the size of the compressed file header (S_H) in bytes is given by:

$$S_H = S_{ACW} + S_F + S_S + s(8 + L) \quad (3)$$

Since both S_{ACW} and S_S are fixed 8-byte fields, (3) can be simplified to:

$$S_H = 16 + S_F + s(8 + L) \quad (4)$$

Thus, as shown in Fig. 2, the size of the compressed file (S_C) can be given by:

$$S_C = S_H + B \quad (5)$$

Where B is a positive integer number represents the number of blocks into which the binary sequence is subdivided. For a binary sequence of N -bit length, B is calculated by:

$$B = \left\lceil \frac{N}{n} \right\rceil \quad (6)$$

If the length of the last block is less than n -bit, then it should be padded with 0s. The number of padding bits (g), which is added to the last block is calculated by:

$$g = B * n - N \quad (7)$$

Substituting (4) and (6) for S_H and B into (5) gives:

$$S_C = 16 + S_F + s(8 + L) + B \quad (8)$$

TABLE I. Compressed file header

| Field | Length | Description |
|----------------|--------|---|
| ACW | 3 | Name of algorithm. |
| V | 1 | Version of the algorithm. |
| F | 1 | Coding format. |
| Char(n) | 1 | The character wordlength. |
| Char(d) | 1 | The number of the different decimal values. |
| Char(g) | 1 | The number of padded bits. |
| D_1 to D_d | L | The actual decimal values of the blocks. |

D. Computing the Compression Ratio

The compression ratio (C) of the ACW(n,s) scheme can be calculated by:

$$C = \frac{S_o}{S_c} = \frac{S_o}{16 + S_F + s(8 + L) + \left\lceil \frac{N}{n} \right\rceil} \quad (9)$$

The above equation shows that C mainly depends on the values of s and n . The compression algorithm starts by selecting a range of n values, then for each value of n , C is calculated. The character wordlength that is selected to compress the source file is the one that achieves the highest compression ratio.

V. THE PROPOSED MODEL

This section presents a description of the proposed Web search engine model. As we have said earlier that the model incorporates two bit-level data compression layers, both installed at the back-end processor, one for index compression and one for query compression, so that the search process can be performed at the compressed index-query level and avoid any decompression and communication activities.

In order to be able to perform the search process at the compressed index-query level, it is important to have a data compression technique that is capable of producing the same pattern for the same character that from both the query and the index. The ACW(n,s) scheme can satisfy this important requirement, therefore, it will be used at the compression layers in the proposed model. Fig. 3 outlines the main components of the proposed models and where the compression layers are located.

The proposed search engine model works as follows: At the back-end processor, after the indexer generates the index, and before sending it to the storage device it keeps it in a temporary memory to apply bit-level compression using the ACW(n,s) scheme, and then sends the compressed index file to the storage device. So that it requires less disk-space enabling more documents to be indexed and accessed in comparatively less CPU time. The scheme creates a compressed file header to store parameters that are relevant to compression process, which mainly include the character-to-binary coding pattern. This header should be stored separately to be accessed by the query compression layer.

On the other hand, the query parser, instead of passing the query to the index file, it passes it to the query compression layer before accessing the index file. In order to produce similar binary pattern for the similar compressed characters from the index and the query, the character-to-binary codes used in converting the index file are passed to be used at the query compression layer. If a match is found the retrieved data is decompressed and passed through the ranker and then through the search engine interface to the end-user.

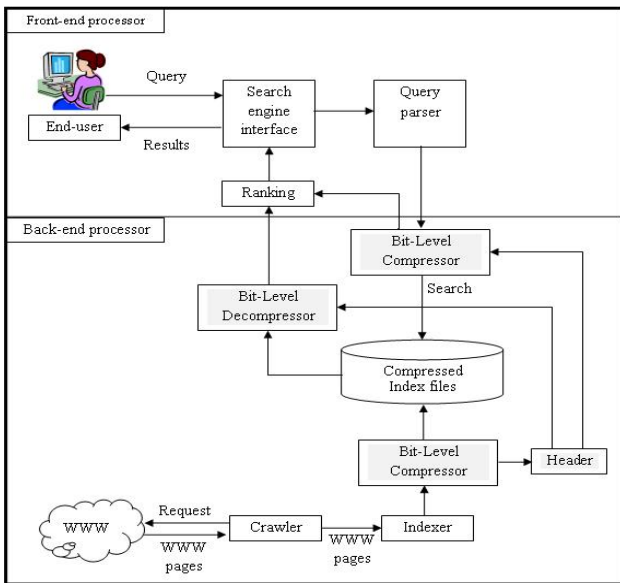


Figure 3. Architecture and main components of proposed search engine model.

VI. EXPERIMENTAL RESULTS

This research is at its early stages and we have not performed any test on compressing a realistic index files. Furthermore, at this stage, little effort has been taken to optimize the runtime of the code, therefore, only results obtained for the compression ratio are presented.

In this paper, we present results for a single experiment to evaluate and compare the performance of the ACW(n,s) scheme. In this experiment, we investigate the variation of C and s with n using two coding formats, namely, the ASCII and the adaptive coding formats. The compression ratios achieved for three text files of various sizes from the Calgary corpus, these are: bib, book1, and paper2 of sizes 11261, 768771, and 821199 Bytes [25]. The results are shown in Table II.

The results showed that: (i) a compression of over 50% can be achieved; (ii) the maximum compression is achieved at $n \leq 14$ depending on the size and the contents of the source file; and (iii) the adaptive coding yields higher compression ratios than the ASCII coding for all values of n .

TABLE II. Variation of C and s with n for the ACW(n,s) scheme.

| n | ASCII coding | | Adaptive coding | |
|---------------------|--------------|-------|-----------------|-------|
| | s | C | s | C |
| bib (111261 Byte) | | | | |
| 9 | 176 | 1.262 | 82 | 1.275 |
| 10 | 214 | 1.394 | 140 | 1.406 |
| 11 | 223 | 1.523 | 174 | 1.537 |
| 12 | 220 | 0.813 | 190 | 0.930 |
| 13 | 211 | 0.701 | 194 | 0.813 |
| 14 | 111 | 1.470 | 111 | 1.477 |
| 15 | 191 | 0.545 | 185 | 0.632 |
| 16 | 181 | 0.499 | 178 | 0.570 |
| book1 (768771 Byte) | | | | |
| 9 | 1056 | 1.266 | 237 | 1.281 |
| 10 | 1352 | 1.397 | 600 | 1.414 |
| 11 | 1447 | 1.531 | 865 | 1.547 |
| 12 | 1448 | 0.898 | 1033 | 1.164 |
| 13 | 1412 | 0.792 | 1135 | 1.064 |
| 14 | 552 | 1.647 | 552 | 1.674 |
| 15 | 1296 | 0.634 | 1185 | 0.853 |
| 16 | 1233 | 0.573 | 1162 | 0.763 |
| paper2 (82199 Byte) | | | | |
| 9 | 113 | 1.265 | 28 | 1.280 |
| 10 | 144 | 1.397 | 68 | 1.413 |
| 11 | 155 | 1.530 | 95 | 1.546 |
| 12 | 155 | 0.846 | 113 | 1.092 |
| 13 | 151 | 0.745 | 123 | 0.997 |
| 14 | 54 | 1.637 | 54 | 1.641 |
| 15 | 138 | 0.599 | 127 | 0.784 |
| 16 | 132 | 0.544 | 124 | 0.701 |

VII. CONCLUSIONS

Although we have not performed realistic tests on truly generated index files, we believe that introducing the new compression layers and utilizing a compression scheme such as the ACW(n,s) scheme enables performing the search process on compressed index-query level, so that less disk space is required for storing index files, increases query throughput and consequently retrieval rate. On the other hand, compressing the search query reduces I/O overheads and query processing time as well as the system response time. The feature of the ACW(n,s) scheme enables performing the search process on different subsequences simultaneously, or searching multiple queries on the same compressed index file.

Since this research is at its early stage, a number of recommendations can be pointed-out for future work, such as: evaluate and compare the performance of the new Web search engine model. Evaluate the performance of the new model Investigate the under simultaneous and multiple query processing.

REFERENCES

- [1] M. Levene. An introduction to search engine and navigation. Pearson Education, 2005.
- [2] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. *Journal of Computer Networks and ISDN Systems*, Vol. 30, No. 1-7, 1998, pp. 107-117.
- [3] T. Calishain. *Web Search Garage*. Prentice-Hall, 2004.
- [4] T. Fagni, R. Perego, F. Silvestri, and S. Orlando. Boosting the performance of Web search engines: Caching and prefetching query results by exploiting historical usage data. *ACM Transactions on Information Systems*, Vol. 24, Issue 1, 2006, pp. 51-78.
- [5] P. Ferragina, F. Luccio, G. Manzini, and S. Muthukrishnan. Structuring labeled trees for optimal succinctness, and beyond. *Proceedings of IEEE Symposium on Foundations of Computer Science (FOCS'05)*, 2005, pp. 184-193.
- [6] R. Gonzalez, and G. Navarro. Statistical encoding of succinct data structures. *Proceeding of the 17th Annual Conference on Combinatorial Pattern Matching (CPM'06)* (Editors: M. Lewenstein and G. Valiente), Barcelona, Spain, 5-7 July 2006, pp. 295-306.
- [7] P. Ferragina and G. Manzini. Indexing compressed texts. *Journal of ACM*, Vol. 52, No. 4, 2005, pp. 552-581.
- [8] P. Ferragina, R. Gonzalez, G. Navarro, and R. Venturini. Compressed text indexes: From theory to practice. *Journal of Experimental Algorithmics (JEA)*, Vol. 13, Section 1, Article No. 12, 2009.
- [9] Z. Chen, J. Gehrke, and F. Korn. Query optimization in compressed database systems. *Proceedings of the Association for Computing Machinery (ACM) – Special Interest on Management of Data (SIGMOD'01) Conference* (Editor: Walid G. Aref), Santa Barbara, California, USA, 21-24 May 2001, pp. 271-282.
- [10] X. Long and T. Suel. Optimized query execution in large search engines with global page ordering. *Proceedings of the 29th International Conference on Very Large Databases (VLDB'03)* (Editors: J. C. Freytag, P. C. Lockemann, S. Abiteboul, M. J. Carey, P. G. Selinger, and A. Heuer), Berlin, Germany, 9-12 September 2003.
- [11] C. Baeza-Yates, B. Ribeiro-Neto, and N. Ziviani. Distributed query processing using partitioned inverted files. *Proceedings of the 9th String Processing and Information Retrieval (SPIRE) Symposium*, September 2002.
- [12] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Computing Surveys*, Vol. 38, No. 2, 2006, pp. 1-56.
- [13] P. Ferragina, G. Manzini, V. Makine, and G. Navarro. Compressed representation of sequences and full-text indexes. *ACM Transactions on Algorithms*, Vol. 3, No. 2, 2007.
- [14] H. Al-Bahadili and S. M. Hussain. A bit-level text compression scheme based on ACW algorithm. *International Journal of Automation and Computing (IJAC)*, Vol. 7, No. 1, 2010, pp. 128-136.
- [15] H. Al-Bahadili and S. M. Hussain. An adaptive character wordlength algorithm for data compression. *Journal of Computers & Mathematics with Applications*, Vol. 55, Issue 6, 2008, pp. 1250-1256.
- [16] E. S. de Moura, G. Navarro, and N. Ziviani. Indexing compressed text. In *proceedings of the 4th South American Workshop on String Processing* (Editor: R. Baeza-Yates), Carleton University Press International Informatics, Canada, 1997, Vol. 8, pp. 95-111.
- [17] S. Varadarajan and T. C. Chiueh. SASE: Implementation of a compressed text search engine. *USENIX Symposium on Internet Technologies and Systems*, 1997.
- [18] R. Grossi, A. Gupta, and J. Vitter. High-order entropy-compressed text indexes. *Proceedings of the ACM SIAM Symposium on Discrete Algorithms (SODA'03)*, 2003, pp. 841-850.
- [19] V. N. Anh and A. Moffat. Index compression using fixed binary codewords. *Proceedings of the 15th Australasian Database Conference* (Editors K. D. Schewe and H. Williams), Dunedin, New Zealand, January 2004.
- [20] R. Gonzalez and G. Navarro. Compressed text indexes with fast locate. *Proceedings of the 18th Annual Symposium on Combinatorial Pattern Matching (CPM'07)* (Editors: B. Ma and K. Zhang), London, Canada, 9-11 July 2007, pp. 216-227.

- [21] R. Gonzalez and G. Navarro. A compressed text index on secondary memory. *Proceedings of the 18th International Workshop on Combinatorial Algorithms (IWOCA'07)*, College Publications, UK, 2007, pp. 80-91.
- [22] A. Moffat, and J. S. Culpepper. Hybrid bitvector index compression. *Proceedings of the 12th Australasian Document Computing Symposium*, Melbourne, Australia December 2007, pp. 25-31.
- [23] S. Melnik, S., Raghavan, B. Yang, and H. Garcia-Molina. Building a distributed full-text index for the Web. *Proceedings of the 10th International World Wide Web Conference*, Hong-Kong, 2-5 May 2000.
- [24] J. S. Vitter. Dynamic Huffman coding. *Journal of ACM*, Vol. 15, No. 2, 1989, pp. 158-167.
- [25] T. C. Bell, J. C., Cleary, and I. H. Witten. *Text Compression*. Prentice-Hall, 1990.



Hussein Al-Bahadili (hbahadili@uop.edu.jo) received his B.Sc degree in Engineering from University of Baghdad in 1986. He received his M.Sc and PhD degrees in Engineering from University of London in 1988 and 1991, respectively. His field of study was parallel computers. He is currently associate professor at Petra University, Amman, Jordan. He is a visiting researcher at the Wireless Networks and Communications Centre (WNCC) at University of Brunel, UK. He is also a visiting researcher at the Centre of Osmosis Research and Applications (CORA), University of Surrey, UK. He has published many papers and book chapters in different fields of science and engineering in numerous leading scholarly and practitioner journals, and presented at leading world-level scholarly conferences. His research interests include parallel and distributed computing, wireless communications, computer networks, cryptography and network security, data compression, image processing, and artificial intelligence and expert systems.



Saif Al-Saab (saif.alsaab@gmail.com) received his B.Sc degree in Engineering from Al-Mustansiriyah University (Baghdad-Iraq) in 1999. He received his M.Sc in Computer Information System from the Arab Academy for Banking & Financial Sciences (Amman-Jordan) in 2003, where he is currently doing his PhD in Computer Information System. He is carrying own his PhD research on the development of a new Compressed Index-query Web Search Engine Model. He has published several papers in different fields of science in numerous leading scholarly and practitioner journals, and presented at leading world-level scholarly conferences. His research interests include search engine technology, data compression, artificial intelligence, and expert systems.