# Comparative Analysis of Sparse Matrix Algorithms For Information Retrieval

Nazli Goharian, Ankit Jain, Qian Sun
Information Retrieval Laboratory
Illinois Institute of Technology
Chicago, Illinois
{goharian,ajain,qian@ir.iit.edu}

## Abstract

We evaluate and compare the storage efficiency of different sparse matrix storage formats as index structure for text collection and their corresponding sparse matrix-vector multiplication algorithm to perform query processing in information retrieval (IR) application. We show the results of our implementations for several sparse matrix algorithms such as Coordinate Storage (COO), Compressed Sparse Column (CSC), Compressed Sparse Row (CSR), and Block Sparse Row (BSR) sparse matrix algorithms, using a standard text collection. Evaluation is based on the storage space requirement for each indexing structure and the efficiency of the query-processing algorithm. Our results demonstrate that CSR is more efficient in terms of storage space requirement and query processing timing over the other sparse matrix algorithms for Information Retrieval application. Furthermore, we experimentally evaluate the mapping of various existing index compression techniques used to compress index in information retrieval systems (IR) on Compressed Sparse Row Information Retrieval (CSR IR).

## 1. Introduction

The mounting available information necessitates the invention of enhanced information retrieval systems. The decisive factors in the performance evaluation of an information retrieval engine are the disk space consumption and query processing time. The terabytes of information available on the Internet poses a grave challenge before experts to investigate into distinctive algorithms that can be used for the purpose and not to mention that information available on the Internet is growing with an explosive rate.

For years inverted index algorithm is treated as the de facto standard for information retrieval systems. Inverted index structure supports a fast query processing. Furthermore it uses compression techniques to achieve a better storage on disk. Inverted index has it's own limitations, such as complexity of update to inverted index and parallelization of inverted index. An alternative approach to inverted index is given in [1], to store the index of text collection in a sparse matrix structure and perform query processing using sparse matrix-vector multiplication. The approach is parallelized and achieved a substantial efficiency over the sequential inverted index [2]. In this paper we investigate the standard BLAS sparse matrix algorithms [3], namely Coordinate Storage (COO), Compressed Sparse Column (CSC), Compressed Sparse Row (CSR) and Block Sparse Row (BSR). In this paper we give a comparative analysis among the sparse matrix algorithms for information retrieval. We compare these storage structures and the efficiency of their multiplication algorithms to perform query processing. Furthermore, we evaluate various compression techniques used to compress inverted index on CSR IR. A comparison of compression ratio and query processing timing of several different conventional index compression schemes is given in [4]. The readers are referred to [5; 6; 7] for information retrieval topics. The remaining of this paper is organized as the following: Section 2 gives a brief introduction of information retrieval engine architecture. Section 3 gives a brief description of each sparse matrix algorithm in the context of an IR application, with the aid of an example. The experiments and the results of experimental evaluation of the implementations and the analysis are given in sections 4 and 5. The results and analysis of index compression techniques on CSR IR are presented in section 6. Finally, we conclude the paper in section 7.

## 2. Architecture

First significant component of an information retrieval system (IR) is the indexing component. Indexing involves creating an index structure that provides fast access to the data for query processing. Figure 1 shows the architecture of such system. The Parser component eliminates the tags and extracts the text to be parsed from the documents. Furthermore, parser eliminates the stop words from the text and passes the text to the indexer. Stop word elimination is a technique to effectively remove the frequently used insignificant words such as *"a"*, *"an"*, *"the"*, to reduce the size of the index. The Indexer component of an IR system then associates a weighing factor defined as inverse document frequency (idf) with each token and calculates term frequency (tf) for each term in a document. (tf is the number of occurrences of a

* This work is supported in part by the National Science Foundation under contract # 0119469.

term in a document; idf is the inverse of document frequency, i.e., an indicator to show the importance of a term, often calculated as log(d/df), where d is the number of documents in collection and df is the number of documents in which a given term appears). Indexer stores the information about every term in every document of the collection in a storage structure called index, so that it can be efficiently accessed at the query time.

The query processor component takes the query from the user, accesses the index to process the query by generating similarity scores for documents that are retrieved for a given query. The similarity scores are then ranked to indicate the relevance of each retrieved document to the user query. The reader is referred to [8] for detailed description of the architecture.

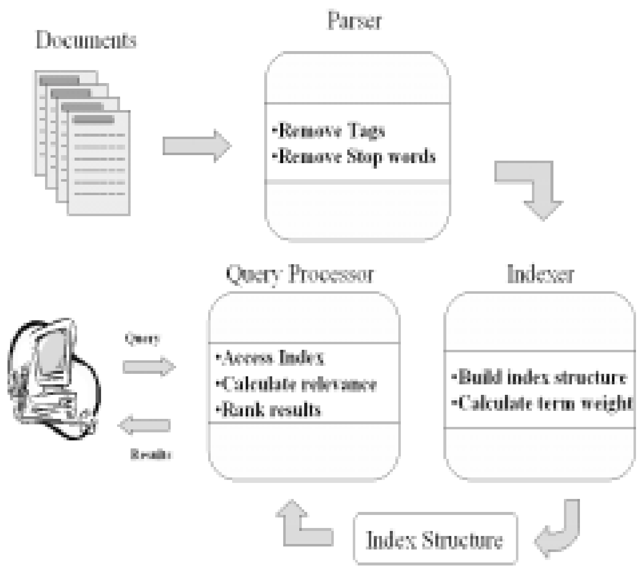

**Figure 1:** Information Retrieval System Architecture

## 3. Algorithms

### 3.1 Coordinate Storage (COO)

A sparse matrix stores only non-zero elements to save space [9]. The simplest sparse matrix storage structure is COO. The index structure is stored in three sparse vectors in COO. The first vector (non-zero vector) stores non-zero elements of the sparse matrix. Non-zero elements of the sparse matrix in information retrieval system correspond to the distinct terms that appear in each document. The term weight (tf-idf) of the term in a particular document represents the importance of a term in a document. Hence we store (tf * idf) for each element in non-zero vector. Second vector in COO is the column vector. Each element of column vector stores the term identifier or the column index for the corresponding term in non-zero vector. The third vector is the row vector that stores the respective document identifier or the row index for each term in the non-zero vector [10].

An example of building index structure is shown with a sample collection in figure 2 with documents D0, D1, D2, D3, D4, and query Q. Table 1 gives the document frequency (df) and term weight (idf) of the terms in the whole sample collection. The matrix representation of the document collection is shown in table 2. Figure 3 is the COO representation of sample collection. The COO sparse matrix-vector multiplication algorithm to perform query processing is shown in figure 4.

Using the algorithm given in figure 4, index structure of figure 3, and the query vector of figure 2, presented as Query Vector: <0.22, 0, 0, 0, 0.4, 0, 0, 0, 0>, the results of query processing and relevance ranking are shown in table3 and table 4.

| | |
|---|---|
| D0: | apple apple eve eve |
| D1: | eve adam eve adam |
| D2: | apple portable computer |
| D3: | big apple new york |
| D4: | fast computer |
| Q: | apple computer |

**Figure 2:** Sample Document Collection and Query

| Term ID | Term | df | Idf |
|---|---|---|---|
| 0 | apple | 3 | 0.22 |
| 1 | eve | 2 | 0.4 |
| 2 | adam | 1 | 0.7 |
| 3 | portable | 1 | 0.7 |
| 4 | computer | 2 | 0.4 |
| 5 | big | 1 | 0.7 |
| 6 | new | 1 | 0.7 |
| 7 | york | 1 | 0.7 |
| 8 | fast | 1 | 0.7 |

**Table 1:** Document frequency (df) and Inverse Document Frequency (Idf) for the Sample Collection

### 3.2 Compressed Sparse Row (CSR)

CSR permits indexed access to rows. Similar to COO, CSR storage structure also consists of three sparse vectors, non-zero vector, column vector and row vector. Index structure differs in the formation of row vector. In CSR row vector consists of pointers to each row of the matrix. The row vector consists of only one element for each row of matrix and the value of element is the position of the first non-zero element of each row in non zero vector. Figure 5 shows CSR storage structure for sample collection shown in figure 2.

| | | **apple** | **eve** | **adam** | **portable** | **computer** | **Big** | **new** | **york** | **fast** |
|---|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| **D0** | 0 | 0.44 | 0.8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **D1** | 1 | 0 | 0.8 | 1.4 | 0 | 0 | 0 | 0 | 0 | 0 |
| **D2** | 2 | 0.22 | 0 | 0 | 0.7 | 0.4 | 0 | 0 | 0 | 0 |
| **D3** | 3 | 0.22 | 0 | 0 | 0 | 0 | 0.7 | 0.7 | 0.7 | 0 |
| **D4** | 4 | 0 | 0 | 0 | 0 | 0.4 | 0 | 0 | 0 | 0.7 |
| **Q** | | 0.22 | 0 | 0 | 0 | 0.4 | 0 | 0 | 0 | 0 |

**Table 2:** Matrix Generated for the Sample Collection

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| non_zero_vector | 0.44 | 0.8 | 0.8 | 1.4 | 0.22 | 0.7 | 0.4 | 0.22 | 0.7 | 0.7 | 0.7 | 0.4 | 0.7 |
| column_vector | 0 | 1 | 1 | 2 | 0 | 3 | 4 | 0 | 5 | 6 | 7 | 4 | 8 |
| row_vector | 0 | 0 | 1 | 1 | 2 | 2 | 2 | 3 | 3 | 3 | 3 | 4 | 4 |

**Figure 3:** COO Storage structure

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| non_zero_vector | 0.44 | 0.8 | 0.8 | 1.4 | 0.22 | 0.7 | 0.4 | 0.22 | 0.7 | 0.7 | 0.7 | 0.4 | 0.7 |
| column_vector | 0 | 1 | 1 | 2 | 0 | 3 | 4 | 0 | 5 | 6 | 7 | 4 | 8 |
| row_vector | 0 | 2 | 4 | 7 | 11 | 13 | | | | | | | |

**Figure 5:** CSR Storage Structure

```
doc_id = 0;
for (count=0; count<M; count++)
    if(doc_id != row_ind[count])
        COO_output[doc_id] = temp;
    doc_id = row_ind[count];
    temp=0;
    endif
    col_ind = col_vector[count];
    temp = temp + non_zero_vector[count] * Q[col_ind];
endfor
Where M is the number of elements in the non_zero vector
```

**Figure 4:** COO Vector Matrix Multiplication Algorithm

| **Document ID** | **Score** |
|---|---|
| D0 | (0.44) * (0.22) +(0.8)*0 = 0.097 |
| D1 | (0.8) * 0 + (1.4) * 0 = 0 |
| D2 | (0.22) * (0.22) + (0.7) * 0 + (0.4) * (0.4) = 0.21 |
| D3 | (0.22) * (0.22) + (0.7) * 0 + (0.7) * 0+ (0.7) * 0 = 0.05 |
| D4 | (0.4) * (0.4) + (0.7) * 0 = 0.16 |

**Table 3:** COO Query Processing and Similarity Scores

| **Document ID** | **Rank** |
|---|---|
| D2 | 1 |
| D4 | 2 |
| D0 | 3 |
| D3 | 4 |

**Table 4:** COO Document Rankings

It can be noticed here that CSR storage structure saves space on disk because unlike COO that stores one entry per non-zero element in row vector, CSR stores only one element for each row of the matrix. The CSR sparse matrix-vector multiplication algorithm to perform query processing is shown in figure 6.

Using the algorithm of figure 6, index structure of figure 5, and the query vector of figure 2, results of query processing and relevance ranking are shown in table 5 and 6.

```
for (count=0; count<M; count++)
    temp=0;
    for(row_ind=row_vector[count];
            row_ind<=(row_vector[count+1]-1);
        row_ind++)
        col_ind = col_vector[row_ind];
        temp = temp + non_zero_vector[row_ind] * Q[col_ind];
    endfor
    CSR_output[count] = temp;
Endfor
Where M = number of documents
```

**Figure 6 :** CSR Vector Matrix Multiplication Algorithm

| **Document ID** | **Score** |
|---|---|
| D0 | (0.44) * (0.22) +(0.8)*0 = 0.097 |
| D1 | (0.8) * 0 + (1.4) * 0 = 0 |
| D2 | (0.22) * (0.22) + (0.7) * 0 + (0.4) * (0.4) = 0.21 |
| D3 | (0.22)*(0.22)+ (0.7)*0+ (0.7)*0+ (0.7)= 0.05 |
| D4 | (0.4) * (0.4) + (0.7) * 0 = 0.16 |

**Table 5:** CSR Query Processing and Similarity Scores

| Document ID | Rank |
|---|---|
| D2 | 1 |
| D4 | 2 |
| D0 | 3 |
| D3 | 4 |

**Table 6:** CSR Document Rankings

## 3.3 Compressed Sparse Column (CSC)

CSC in deviation to CSR and COO permits indexed access to column of the matrix. Similar to COO and CSR, CSC storage structure also consists of three sparse vectors, non-zero vector, column vector and row vector. Non-zero vector stores the non-zero elements of each column of the matrix and column vector stores pointer to the first non-zero element of each column. Row vector stores the row index associated with each non-zero element. Figure 7 shows storage structure for CSC for sample collection shown in table 1. The CSC sparse matrix-vector multiplication algorithm to perform query processing is shown in figure 8.

Using the algorithm given in figure 8, index structure of figure 7, and the query vector of figure 2, the results of query processing and relevance ranking are shown in table 7 and 8.

| | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| non_zero_vector | 0.44 | 0.22 | 0.22 | 0.8 | 0.8 | 1.4 | 0.7 | 0.4 | 0.4 | 0.7 | 0.7 | 0.7 | 0.7 |
| column_vector | 0 | 3 | 5 | 6 | 7 | 9 | 10 | 11 | 12 | 13 | | | |
| row_vector | 0 | 2 | 3 | 0 | 1 | 1 | 2 | 2 | 4 | 3 | 3 | 3 | 4 |

**Figure 7:** CSC Storage Structure

| | | **Apple** | **eve** | **adam** | **Portable** | **computer** | **big** | **new** | **york** | **fast** | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| **D0** | 0 | 0.44 | 0.8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **D1** | 1 | 0 | 0.8 | 1.4 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **D2** | 2 | 0.22 | 0 | 0 | 0.7 | 0.4 | 0 | 0 | 0 | 0 | 0 |
| **D3** | 3 | 0.22 | 0 | 0 | 0 | 0 | 0.7 | 0.7 | 0.7 | 0 | 0 |
| **D4** | 4 | 0 | 0 | 0 | 0 | 0.4 | 0 | 0 | 0 | 0.7 | 0 |
| | 5 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

**Table 9:** Modified Matrix Generated for the Sample collection for BSR

| | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| non_zero_vector | 0.44 | 0.8 | 0 | 0 | 0.22 | 0 | 0 | 0.7 | 0.4 | 0 | 0 | 0 | 0.4 | 0 | 0.7 | 0 |
| | 0 | 0.8 | 1.4 | 0 | 0.22 | 0 | 0 | 0 | 0 | 0.7 | 0.7 | 0.7 | 0 | 0 | 0 | 0 |
| column_vector | 0 | 2 | 0 | 2 | 4 | 6 | 4 | 8 | | | | | | | | |
| Row_vector | 0 | 2 | 6 | 8 | | | | | | | | | | | | |

**Figure 9:** BSR Storage Structure

```
for (count=0; count<M; count++)
        for(col_ind=col_vector[count];
          col_ind<=(col_vector[count+1]-1);col_ind++)
        row_ind =row_vector[col_ind];
        CSC_output[row_ind] = CSC_output[row_ind] +
          non_zero_vector[col_ind] * Q[col_ind];
    endfor
endfor

Where M = number of distinct terms
```

**Figure 8:** CSC Vector Matrix Multiplication Algorithm

| **Calculation** | **Document Score** |
|---|---|
| 0.44 * 0.22 = 0.097 | D0 = 0.097 |
| 0.22 * 0.22 = 0.05 | D2 = 0.05 |
| 0.22 * 0.22 = 0.05 | D3 = 0.05 |
| 0.8 * 0 = 0 | D0 = 0.097 |
| 0.8 * 0 = 0 | D1 = 0 |
| 1.4 * 0 = 0 | D1 = 0 |
| 0.7 * 0 = 0 | D2 = 0.05 |
| 0.4 * 0.4 = .16 | D2 = 0.05 + 0.16 = 0.21 |
| 0.4 * 0.4 = .16 | D4 = 0.16 |
| 0.7 * 0 = 0 | D3 = 0.05 |
| 0.7 * 0 = 0 | D3 = 0.05 |
| 0.7 * 0 = 0 | D3 = 0.05 |
| 0.7 * 0 = 0 | D4 = 0.16 |

**Table 7:** CSC Query Processing and Similarity Scores

| Document ID | Rank |
|---|---|
| D2 | 1 |
| D4 | 2 |
| D0 | 3 |
| D3 | 4 |

**Table 8:** CSC Document Rankings

## 3.4 Block Sparse Row (BSR)

BSR is different from the other three algorithms that we discussed so far. Each element of non-zero vector in BSR is mapped to a non-zero square block of *n* dimensions. The block row algorithm assumes that the number of non-zero elements in each row is a multiple of block size. Additional zeros are stored in a block to satisfy this condition. In BSR a non-zero vector is a rectangular array that stores non-zero blocks in row fashion, column vector stores the column indices of the first element of each non-zero block and row vector stores the pointer to each block row in the matrix. Figure 9 shows storage structure for BSR for sample collection shown in table 1, taking a block size of 2. As earlier discussed that additional zeros can be added to make n dimensional blocks, we added a dummy document as the last row and a dummy term as the last column with all zero elements to the initial matrix shown in table 2. The modified matrix is presented in table 9. The BSR sparse matrix-vector multiplication algorithm to perform query processing is shown in figure 10. We consider a 2x2 block size. Using the algorithm given in figure 10, index structure of figure 9, and the query vector of figure 2, the results of query processing and relevance ranking are shown in tables 10 and 11.

```
for(count = 0; count < M; count++)
  doc_n = 2 * count;
  temp = 0;
  for(row_ind = row[count]; row_ind <= row[count+1] -1;
                                          row_ind++)
     col_ind = col[row_ind];
     non_ind = row_ind * 2;
     temp = Q[col_ind];
     if (temp > 0)
        BSR_output[doc_n] =  BSR_output[doc_n]
                          + non_zero[0,non_ind] * temp;
        BSC_output [doc_n+1] = BSR_output [doc_n+1]
                          +  non_zero[1,non_ind] * temp;
        temp = Q[col_ind+1];
     endif
     if (temp > 0)
        BSC_output [doc_n] = BSR_output [doc_n]
                     + non_zero[0,non_ind + 1] * temp;
        BSC_output [doc_n+1]= BSR_output [doc_n+1]
                     + non_zero[1,non_ind + 1] * temp;
     endif
  endfor
endfor
```

*Where M = number of documents/2 or (number of document+ 1)/2 if number of documents is odd*

**Figure 10:** BSR Vector Matrix Multiplication Algorithm

| Calculation | Document Score |
|---|---|
| 0.44 * 0.22 = 0.02 | D0 = 0.097 |
| 0 * 0.22 = 0 | D1 = 0 |
| 0.22 * 0.22 = 0.05 | D2 = 0.05 |
| 0.22 * 0.22 = 0.05 | D3 = 0.05 |
| 0.05 + 0.4 * 0.4 = 0.21 | D2 = 0.21 |
| 0.05 + 0 * 0.4 = 0.05 | D3 = 0.05 |
| 0.4* 0.4 = 0.16 | D4 = 0.16 |

**Table 10:** BSR Query Processing and Similarity Scores

| Document ID | Rank |
|---|---|
| D2 | 1 |
| D4 | 2 |
| D0 | 3 |
| D3 | 4 |

**Table 11:** BSR Document Rankings

## 4. Experimental Results of Sparse Matrix Algorithms for IR

We performed our experiments with a standard benchmark of text collection provided by Text Retrieval Evaluation Conference (TREC) sponsored by National Institute of Standard and Technology (NIST) [11]. We use TREC disks 4 and 5 data, a 2 Gigabytes text collection. We choose 50 TREC topic and descriptive queries to compare the query processing efficiency of the sparse matrix algorithms. Topics are the small queries each having 1 to 4 words. Descriptive queries are usually longer queries, each with 5 to 30 words. The benchmark text collection statistics are in table 12.

| | |
|---|---|
| Document Collection Size | 2GB |
| Number of Files Parsed | 2,295 |
| Number of Documents Parsed | 527,580 |
| Total Number of Terms in Collection (distinct in each document & excluding Stop Terms) | 77,234,607 |
| Number of Distinct Terms in Collection (excluding Stop Terms) | 994,243 |

**Table 12:** TREC disks 4-5 Text Collection Statistics

The experiments are performed on a 1 GHz, 4 GB RAM, Sun ES 450 server. For each of sparse matrix storage structures, described on the previous sections and used to implement our retrieval engine prototype, the disk storage requirement is measured and provided in table 13 and figure 11. The experimental results demonstrate that CSR storage structure takes the least amount of disk space compare to the other three structures.

To study the query processing time, we used 50 TREC *topic* and *descriptive* queries on our collection using each storage structure and the corresponding sparse matrix vector multiplication to perform query processing. The average query processing time for each set of 50 queries for each approach is recorded. The results for *topics* and *descriptive* queries are shown in figures 12 and 13.

| Storage Formats | Disk Space Consumption in Megabytes |
|---|---|
| COO | 1541 |
| CSR | 1021 |
| CSC | 1147 |
| BSR | 1581 |

**Table 13:** Disk Space Requirements for CSR, CSC, COO, and BSR Sparse Matrix Structures for TREC disks 4-5 Text Collection

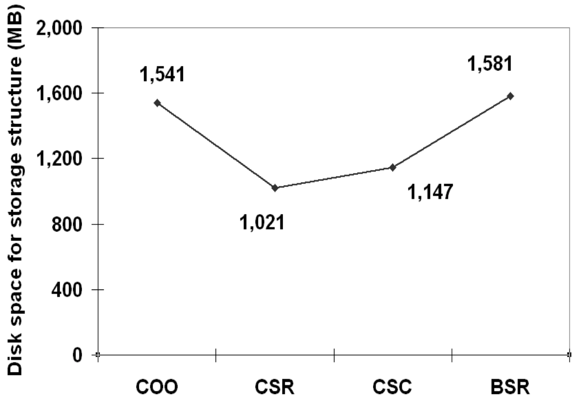

**Figure 11:** Disk Space Requirements for CSR, CSC, COO, and BSR Sparse Matrix Structures for TREC disks 4 -5 Text Collection

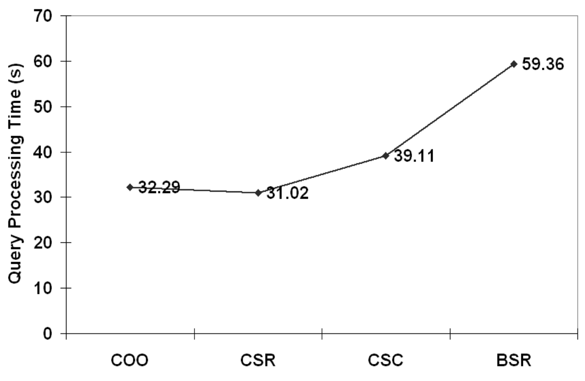

**Figure 12:** Query Processing Time for each Algorithm using 50 TREC Topic Queries

As shown in figure 12 and figure 13, in both cases of topic and descriptive, i.e. short and medium size of queries, the Compressed Sparse Row (CSR) performs the best in terms of query processing timing.

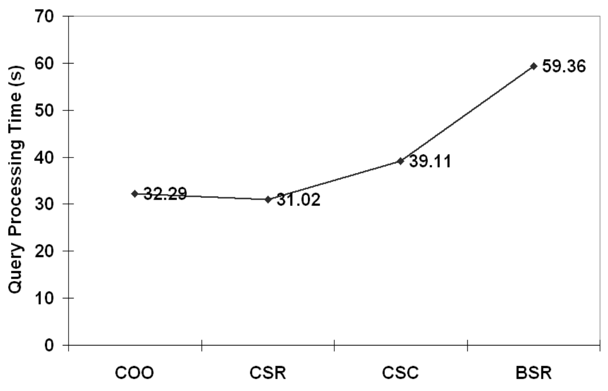

**Figure 13:** Query Processing time for each Algorithm for 50 TREC Descriptive Queries

| TREC disks 4-5 | Number of Elements in Non-Zero Vector | Number of Elements in Column Vector | Number of Elements i Row Vector |
|---|---|---|---|
| **CSR** | 77,234,607 | 77,234,607 | 527,580 |
| **CSC** | 77,234,607 | 994,243 | 77,234,607 |
| **COO** | 77,234,607 | 77,234,607 | 77,234,607 |
| **BSR (2x2 blk)** | 298,622,476 | 74,655,619 | 263,790 |

**Table 14:** Number of TREC Data Elements in each Sparse Matrix Storage Formats

## 5. Analysis

The sparse matrix storage formats, COO, CSR and CSC, are quite similar to each other. All three formats store the total number of non-zero elements, i.e., total number of terms in the collection calculated based on the total number of unique terms of each document in collection. The difference among these three structure formats is in column and row vectors. COO stores row and column indices for each non-zero element that makes it the least efficient structure when using it for text collection domain as compare to CSR and CSC. CSR in deviation to COO and CSC stores only the pointers to each row in row vector and this marks the improvement in the storage in CSR over both COO and CSC, since we store only one element per document in row vector. As shown in table 14, the row vector of CSR is about 78 times smaller than the row vector of COO and CSC for our 2 GB TREC text collection. The fact that average document length of TREC documents is approximately 80-100 words, explains the numbers. For column vector, CSC stores the pointers to the columns and that means only one value in column vector for each unique term as compared with COO and CSR that store the number of non-zero elements in their column vectors.

Table 14 shows that CSC storage will have a definite advantage over COO, but loose to CSR, because of less compression achieved in column vector of CSC than compression in row vector of CSR. BSR consists of blocks that include non-zero elements, along with zero elements. A zero block size BSR will result in a similar structure as COO. In order to make square blocks in BSR, we end up in storing as well zero elements in blocks. The two-dimensional blocks in BSR contain many zeros. Thus, the BSR storage structure takes much larger disk space compare to the other structures we discussed. The larger the dimension of the block in BSR, the higher is the number of zeros to be stored, which consequently requires more disk space.

To process a query, we have to traverse through all the elements of non-zero vector, column vector and row vector to calculate the score of all the documents and then sort the documents on their relevance scores to get the top relevant documents. This signifies that the query processing time will also be proportional to the size of storage structure i.e. the more number of elements are stored in the index, the more elements we have to process and consequently the more time is needed for query processing. Based on our analytical and experimental results, the CSR format is identified the best storage format for storing the index of text collection and perform query processing when implementing information retrieval system as the application of sparse matrix vector multiplication.

## 6. Experimental Results of Index Compression Techniques on CSR IR

We mapped several of the index compression techniques that are used to compress an inverted index on Compressed Sparse Row Information Retrieval (CSR IR). Among these methods are Flat Huffman compression, Byte-Aligned, Interpolative, Elias Gamma and Golomb compression schemes. The implementation detail for each of these techniques on CSR IR can be found in [12]. In this section we give a brief description of each technique, followed by our experimental results.

In a ***Flat Huffman*** compression scheme [13; 14], to encode a number in the range 0 to *n* takes $\lceil \log n \rceil$ bits. That is a straight binary encoding. The last bit is not always necessary. To encode without wasting bits, a flat encoding can be used. It is equivalent to having a Huffman tree with all of the leaves within one level of each other.

***Byte Aligned*** compression scheme [6] is a fixed length encoding technique that creates a byte boundary to represent the encoded integer to achieve a faster access. As we noticed that the gap calculated between every two term identifiers belonging to a given document in on our collection is less than $2^{15}-1$, thus each integer needs a maximum of 2 bytes to be encoded. In our implementation of this approach we used blocks of 7 bits, plus an additional bit to indicate if an additional byte is needed. For example, the integer 2 would be encoded as 0 0000010. The first bit indicates that no additional byte is followed.

***Elias Gamma*** compression scheme [6] represents an integer x for $\lfloor \log x \rfloor$ as unary, followed by a zero marker and $\lfloor \log x \rfloor$ bits for the remainder of $x-2^{\lfloor \log x \rfloor}$. The number of the bits in the unary part indicates the number of the bits needed to code the number. For example integer 7 is encoded as 11011.

***Golomb*** compression scheme [15] assigns a parameter *b*, which is to represent the approximate distribution of the values, and encodes an integer x in two parts. The first part is $\left\lfloor \frac{x-1}{b} \right\rfloor$ coded in unary; the remainder $r = x - \left\lfloor \frac{x-1}{b} \right\rfloor b - 1$ is coded in binary with $\lfloor \log b \rfloor$ or $\lceil \log b \rceil$ bits. For x = 9 and b = 3, the integer 9 is encoded in 11011. The value *b* can be calculated in different ways. In our implementations, we calculated $b = \frac{\ln 2}{p}$, where *p* is the probability of the item occurring in a given position. If the item occurs in *a* out of *b* places, $p = \frac{a}{b}$, it amounts to dividing *n* by *b*. The quotient is encoded in unary, and the remainder is encoded with a flat Huffman tree.

***Interpolative*** compression scheme [15] treats the array of indices of the occurrences as a binary search tree. It performs a preorder traversal. At each step, the range of the index is further limited.

Figure 14 shows the comparison of index space requirement for CSR IR before and after mapping various conventional index compression techniques using the 2 GB TREC data collection. As mentioned earlier we used Interpolative, Golomb, Gamma, and Byte-Aligned index compression techniques on CSR IR. We performed combination of compressions on the term identifier and term frequency. For example *Golomb-Gamma* indicates that the term identifier is compressed using Golomb compression and term frequency is compressed using Gamma compression scheme. We performed our experimentations on the following combinations of schemes on the term identifier and term frequency: *Interpolative-Huffman*, *Golomb-Gamma*, *Byte Aligned-Byte*, and *Gamma-Gamma*. The size of the index compressed with the Interpolative-Huffman scheme is 23% of the raw index. This number is 29% for Golomb-Gamma, 28% for Gamma-Gamma, and 38% for the Byte-Aligned scheme. In Figure 15, we illustrate the query processing timing comparisons for all these cases. The results indicate that when the index compressed using the fixed length compression technique, such as Byte-Aligned that has the byte boundaries, it provides a much faster access and thus, reduces the query processing timing.

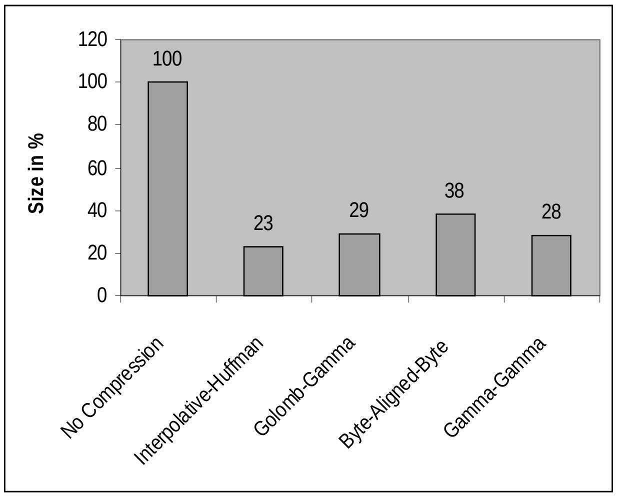

**Figure 14:** Index Size Comparison in Percentage in CSR IR

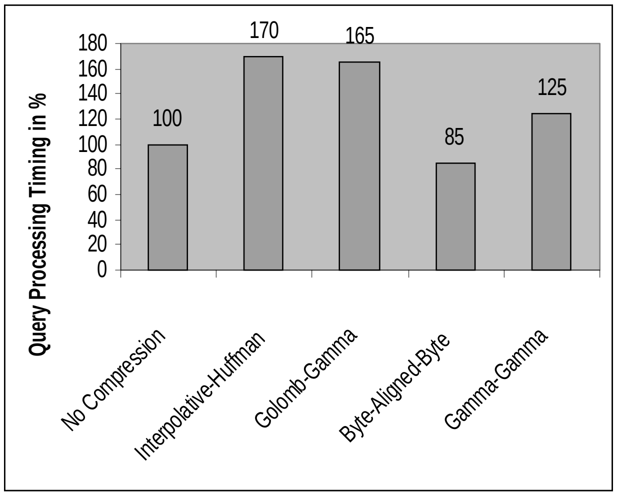

**Figure 15:** Query Processing Timing Percentage using Uncompressed and Compressed Index in CSR IR

## 7. Conclusion

We demonstrated the results of our comparative analysis and experiments on Compressed Column (CSC), Compressed Row (CSR), Column Coordinate (COO), and Block Sparse Row (BSR) sparse matrix algorithms for text information retrieval applications. Our results indicate that CSR storage structure takes the least storage space on the disk and performs the best for the query processing in comparison to COO, CSC and BSR, using 2 GB TREC standard benchmark text collection. Furthermore, We demonstrated experimentally the mapping of the existent compression schemes applied on the inverted index in information retrieval (IR) onto the CSR IR. We used several of the known compression schemes used to compress the inverted index such as Byte Alinged, Golomb, Gamma and Interpolative compression for our experiments. We noticed that same as in the inverted index, a good compression ratio can be achieved by mapping the same compression schemes onto the CSR IR. As far as the query processing timing, depending on the compression technique, the efficiency might decrease or increase.

## References

**[1]** N. Goharian, T. El-Ghazawi, D. Grossman,“Enterprise Text Processing: A Sparse Matrix Approach”, *IEEE International Conference on Information Techniques on: Coding & Computing (ITCC 2001),* Las Vegas, Nevada April 2001.

**[2]** A. Jain, N. Goharian, “On Parallel Implementation of Sparse Matrix Information Retrieval Engine”, *The 2002 International Multi-conferences in Computer Science: on Information and Knowledge Engineering (IKE'02),* Las Vegas, Nevada, June 2002.

**[3]** BLAST Forum, “Documentation for the Basic Linear Algebra Subprograms”, *http://www.netlib.org/blast/blast-forum*, 1999.

**[4]** F. Scholer, H. Williams, J. Yiannis, J. Zobel, “Compression of Inverted Indexes For Fast Query Evaluation”, *proceedings of ACM SIGIR*, 2002.

**[5]** G. Salton, “Automatic Text Processing”, *Addison Wesley*, Massachusetts, 1989.

**[6]** D. Grossman and O. Frieder, “Information Retrieval: Algorithm and Heuristics”, *Kluwer Academic Publishers*, 1998.

**[7]** ACM Special Interest Group in Information Retrieval conference Proceedings, *http://sigir.acm.org*

**[8]** D. Grossman and O. Frieder, "Anatomy of a Search Engine: A Java-based Introduction to Scalable Information Retrieval", manuscript 2002.

**[9]** Sergio Pissanetsky, "Sparse Matrix Technology", *Academic Press*, London, 1984.

**[10]** Nawaaz Ahmed, Nikolay Mateev, Keshav Pingali, and Paul Stodghill. "A Framework for Sparse Matrix Code Synthesis from High-level Specifications", *SC2000*, Dallas, TX, November 2000.

**[11]** Text Retrieval Conference, *http://trec.nist.gov*

**[12]** S. Stein, N. Goharian, "On the Mapping of Index Compression Techniques on CSR Information retrieval", *IEEE International Conference on Computing and Coding (ITCC'03)*, April 2003.

**[13]** D. A. Huffman, "A Method for the Construction of Minimum Redundancy Codes", *Proceedings of the Institute of Radio Engineers*, 40 (1951), 1098-1101.

**[14]** Thomas H. Cormen, et al., "Introduction to Algorithms", 2nd edition, *McGraw Hill Publisher*, 2001.

**[15**] Witten, Moffat, and Bell, "Managing Gigabytes", *Morgan Kaufman Publishers*, 1999.