Enterprise Text Processing: A Sparse Matrix Approach

Nazli Goharian
Illinois Institute of Technology
Chicago, Illinois
nazli@ir.iit.edu

Tarek El-Ghazawi
George Mason University
Fairfax, Virginia
tarek@gmu.edu

David Grossman
Illinois Institute of Technology
Chicago, Illinois
grossman@ir.iit.edu

Abstract

Documents, both internal and related publicly available, are now considered a corporate asset. The potential to efficiently and accurately search such documents is of great significance. We demonstrate the application of sparse matrix-vector multiplication algorithms for text storage and retrieval as a means of supporting efficient and accurate text processing.

As many parallel sparse matrix-vector multiplication algorithms exist, such an information retrieval approach lends itself to parallelism. This enables us to attack the problem of parallel information retrieval, which has resisted good scalability. We use sparse matrix compression algorithms and compare the storage of a sub-collection of the commonly used NIST TREC corpus with a traditional inverted index. We demonstrate query processing using sparse matrix-vector multiplication algorithm. Our results indicate that our approach saves approximately 35% of the total storage requirements for the inverted index. Additionally, to improve accuracy, we develop a novel matrix based, relevance feedback technique as well as a proximity search algorithm. The results of our experiment to incorporate Proximity Search capability into the system also indicate 35% less storage for sparse matrix over inverted index.

1. Introduction

The World Wide Web is doubling nearly every year. Additionally, the amount of electronically available information continues to increase on large intranets. Although the availability of secondary storage continues to rise, the amount of information is, at the least, keeping pace. A book on large-scale systems says:

> *"It might be argued that as machine performance increases, so too will the amount of memory supported, but it seems likely that collection sizes will also grow at the same exponential rate, and that memory will never overtake demand."* [Witten, et al., 1994]

Hence, we can expect the need for better compression algorithms will continue to grow. Additionally, as the intranets inside of large Fortune 500 corporations grow, the need to search and quickly find information will rise.

1.1. General Sparse Matrix Background

Matrices are used in scientific applications to store data but have yet to be efficiently used in the field of information retrieval systems. The nature of text data in a matrix format results in a very highly sparse matrix. To save space and run time, it is important to store only the non-zero elements. There are a variety of compression formats to compress a sparse matrix in a way that it stores only non-zero elements. Among these compression formats is Scalar ITPACK [Peters, 1991 and Petition, 1993]. BLAS (Basic Linear Algebra Subprograms) Technical Forum suggests sparse matrix formats such as Coordinate (COO), Compressed Sparse Row (CSR), Compressed Sparse Column (CSC), Sparse Diagonal (DIA), Block Coordinate (BCO), Block Compressed Sparse Row (BSR), Block Compressed Sparse Column (BSC), Block Sparse Diagonal (BDI), Variable Block Compressed Sparse Row (VBR) [BLAST, 1999].

The Compressed Sparse Row (CSR) format saves some of the storage overhead by storing each row as the pair of non-zero element and column indices. Compressed Sparse Row is general and relatively efficient, hence, we chose this format for our experiments.

The CSR compression stores the compressed sparse in three vectors. The first vector stores the non-zero elements of the sparse matrix. The second vector stores the column indices belonging to the non-zero elements. The elements of the third row, row vector, correspond to the i-th non-zero value of the sparse matrix belonging to the first non-zero element of each row. The offset between every two adjacent value of the row vector is the number of elements that belong to the same row. The last element in the row vector is the value of the last element of row vector incremented by the number of non-zero elements in the last row of sparse matrix. The size of the row vector is $M+1$, M being the number of the rows in the sparse matrix.

1.2. Overview of the Information Retrieval Matrix Approach

In our prior work [Goharian, et.al., 1999, 2000], we demonstrated the application of Sparse Matrix-Vector Multiplication in an Information Retrieval (IR) System. The motivation of our work is to utilize other techniques and codes to implement a scalable IR system. Thus, minimizing the need for the redevelopment of software. Our approach relies on the Vector Space model to compute relevance. We showed the sparse matrix storage method as an alternative to store the inverted index and demonstrated how to map the documents into a matrix. We demonstrated the use of sparse matrix-vector multiplication algorithm to perform query processing and relevance ranking. Figure 4 is the algorithm for CSR sparse matrix-vector multiplication, which is one of the commonly used sparse matrix-vector multiplication algorithms for random pattern sparse matrices. The approach is shown via an example with the sample collection of figure 1.

D0 = matrix matrix compression compression
D1 = matrix compression matrix compression
D2 = vector vector multiplication
D3 = sparse matrix multiplication
Q = matrix compression

Figure 1: Sample collection and query

The collection is parsed and the index of table 1 and table 2 are created. The unique single-terms along with each term's term frequency (*tf*) in a given document, the number of documents in the collection having a given term (*df*), and the importance of the term in the collection, i.e., Inverse Document Frequency (*idf*) are identified. The *idf* is commonly defined as $log(\frac{d}{df})$ where d is the number of documents in the collection. The collection representation in Compressed Sparse Row matrix is shown in figure 2. The elements of the non-zero vector are the weights of terms calculated as *tf*idf* of the terms.

Using the algorithm of figure 4, the query processing is performed on the query of figure 1 and the compressed matrix of figure 2. The query is presented as a vector in figure 3. The non-zero elements of query vector are *tf*idf* of query terms. The result of the query processing and relevance ranking is shown in figure 5. The result shows that the documents D0 and D1 are ranked the highest and after that document D3. The document D2 is not relevant.

Table 1: Term Frequency for sample collection

DOCS	**Tf**
D0	
matrix	2
compression	2
D1	
matrix	2
compression	2
D2	
vector	2
multiplication	1
D3	
sparse	1
matrix	1
multiplication	1

Table 2: df and idf for sample collection

Term_id	**Term**	**Df**	**Idf**
0	compression	2	0.30
1	matrix	3	0.12
2	vector	1	0.60
3	multiplication	2	0.30
4	sparse	1	0.60

```
non_zero_vector = <0.60   0.24   0.60   0.24   1.20   0.30   0.12   0.30   0.60  >
col_vector  =           <0      1        0       1      2       3       1       3       4  >
row_vector =            < 0      2       4       6      9>
```

Figure 2: CSR representation of data

```
Q= <     0.30      0.12      0          0          0      >
```

Figure 3: Query Vector

```
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
```

Figure 4: CSR Sparse Matrix-Vector Multiplication Algorithm

```
DOC[0] = 0+(0.60*0.30)=0.18
DOC[0] = 0.18+(0.24*0.12)=0.21
DOC[1] = 0+(0.60*0.30)=0.18
DOC[1] = 0.18+(0.24*0.12)=0.21
DOC[2] = 0+(1.20*0.00)=0.00
DOC[2] = 0.00+(0.30*0)=0.00
DOC[3] = 0+(0.12*0.12)=0.01
DOC[3] = 0.01+(0.30*0)=0.01
DOC[3] = 0.01+(0.60*0.00)=0.01
```

Figure 5: Result of the relevance ranking

1.3. Proximity Processing

We enhanced our experiment to include the term offsets in each document to implement Proximity Search capability of Information Retrieval both on inverted index and sparse matrix.

Proximity searches are used in Information Retrieval to increase the accuracy of the search by considering a particular query term sequence in the document [Goldman et al, 1998]. The documents, in which the query terms appear within a specific window size, are retrieved and ranked higher than any document that simply contains the query term. For example, the query "information retrieval", with query condition of window size 1, does not rank as high the documents that have the terms "information" and "retrieval" in a sequence with a negative window size such as "Retrieval of Information", or a window size bigger than 1 such as "Information System for Retrieval of Employee Data". An example is the implementation of the Proximity Search on the King James Bible by the University of Michigan. The system searches for a text that includes the search terms specified to be Near, Not Near, Followed By, Not Followed By each other within 40, 80 or 120 characters [UMich, 1997].

We modified the sparse matrix storage structure to implement the Proximity Search. We add a fourth and fifth vector, namely, *offset_vector* and *offset_marker* to the structure. Figure 6 shows the modified structure. The *offset_vector* contains the offset of each term in each given document. The number of elements in the *offset_vector* is total number of all occurrences of unique terms in the collection. The elements in the *offset_marker* indicate the number of the occurrences of each term in a document; hence it shows which offsets in the *offset_vector* belong to a given term in a document. The number of elements in the *offset_marker* is the number of non-zero elements+1. The position of each element in the *offset_marker* corresponds to the position of the term id of the term in *col_vector*, whose offsets are identified in *offset_vector*. The value in *offset_marker* corresponds to the location of the first term offset of a term in the *offset_vector*. We show our structure for the same sample collection in figure 6, and for the query in figure 7. We also modified the CSR matrix-vector multiplication algorithm of figure 4 to be able to perform Proximity search. We proposed our algorithm in our prior work and demonstrated the Proximity Search and relevance ranking on the sample collection.

```
non_zero_vector = <0.60   0.24   0.60   0.24   1.20   0.30   0.12   0.30   0.60  >
col_vector  =        <0       1       0       1       2       3       1       3       4  >
row_vector =         < 0        2        4        6       9>
offset _vector =     < 2   3   0   1   1   3   0   2   0   1  2   1  2   0>
offset_marker=        < 0       2       4       6       8       10     11    12      13>
```

Figure 6: Modified Compressed Sparse Matrix for Proximity Search

```
Q= <0.30  0.12    0     0     0  >
v2=<1      0        0     0     0 >
```

Figure 7: Modified Query for Proximity Search

2. Experimental Results

In the prior work, we showed the analytical results of the storage space for TREC text collection both as Inverted Index and Sparse Matrix structure. The analysis demonstrated that Compressed Sparse Row sparse matrix structure saves 35% - 40% storage space over storage space used in the Inverted Index structure both with and without offset structure for Proximity search. Later experimental results, presented in this paper, prove the initial analysis.

We used a sub-collection of TREC data for our experiment. Table 3 shows the number of documents in the collection, distinct terms and total number of terms in the collection uniquely in each document. The total number of terms in collection uniquely in each document corresponds to the non-zero elements in sparse matrix and to the posting list entries in the Inverted Index. We stored this data in an Inverted Index and measured the storage space. Also, we stored the data in Compressed Sparse Row format of sparse matrix and measured the storage space. A sample document from TREC data collection is shown below:

<HEADLINE>
<P>
LIFE ON EARTH
</P>
</HEADLINE>
<TEXT>
<P>
In response to Lee Dye's article "Galileo Views Earth With an Alien's Eye"
(front page, Dec. 20):
</P>
<P>
In light of all the endless conflicts between men across our planet, Galileo,
the spacecraft, seems to have discerned the root of the problem -- "it detected
no clear sign of intelligent life" on planet Earth!
</P>
<P>
MARILYN E. WHITAKER, Glendale
</P>
</TEXT>
<TYPE>
<P>
Letter to the Editor
</P>

2.1. Inverted Index Implementation

To eliminate unnecessary I/O resulting from the retrieval of non-relevant blocks, text searches often rely on inverted index files [Stone, 1987]. For each key term in the collection, a list of the documents that the term appears in is associated. Each query term is examined against the terms in the index and in the case of a match, the posting list is returned. The set of posting lists is then intersected and the documents containing all the requested terms are returned.

The storage space for the conventional inverted index has two components. The Index component stores the unique terms in the collection, each pointing to the Posting List. The Posting list is the list of all documents having a given term. The storage needed for the posting lists of inverted index tend to grow fast as a new document is encountered having the term. The Efficiency issues and considerations in handling Inverted Indices are discussed in [Frieder, et al., 2000].

The storage space for the Index component in our experiment is compound of storage for the term, document frequency (df) and the square of the Inverse Document Frequency (idf^2). The posting list storage is compound of document identifier and term frequency (*tf*). Our experiment showed that the space taken for storing data in inverted index was 1,888,308 bytes.

Table 3: Experimental result for Storage of Inverted Index and CSR Matrix

Documents	Distinct Terms	NZ Elements	Inverted Index (byte)	Sparse Matrix (byte)
1,828	23,744	320,135	1,888,308	1,199,508

Table 4: Experimental result for Storage of Inverted Index and CSR Matrix for Proximity Search

Documents	Distinct Terms	NZ Elements	Inverted Index (byte)	Sparse Matrix (byte)
1,788	20,580	316,065	5,305,492	3,468,994

2.2. Sparse Matrix Implementation

We stored the $tf*idf^2$ of each collection term as an element in the non-zero vector. We stored the term identifier of each corresponding term in the column vector. The third vector stored the document identifier for the corresponding terms. Storing the square of Inverse Document Frequency was done to reduce the query processing time to find the *idf* of the term. The storage space for our collection indicated the space taken to store the collection was 1,199,508 bytes for the same collection. The results are shown in table 3.

In our implementation for Proximity Search, we modified the *offset_marker* to store term frequency, *tf*, of the terms to save more space. The term frequency stored in *offset_marker* indicates the number of elements stored in *offset_vector* associated to the term identifier, its location in *col_vector* is identified by the location of the element in the *offset_marker*. Thus, the *offset_marker* for the sample collection implemented as < 2 2 2 2 2 1 1 1 1>.

Table 4 shows the result of our experiments to store the collection data with the term offsets for proximity search, both as inverted index and sparse matrix.

3. Conclusion and Directions for Future Work

Previously, we built an analytical model of the Sparse Matrix approach to Information Retrieval [Goharian et al., 1999, 2000]. In this paper, we built an experimental prototype to validate this analysis. The storage reduction of compressed sparse row matrix is 35%-40% over the storage of conventional inverted index. Furthermore, we also experimentally evaluated our proposed Proximity Search structure to improve the accuracy of relevance ranking. The results for Proximity Search also prove our analytical results of the previous effort and reduce also 35%-40% of the storage in compressed sparse row matrix over the inverted index. In the future, we will evaluate the approach on a parallel platform.

4. References

[BLAST, 1999] BLAST Forum, http://www.netlib.org/blast/blast-forum, 1999.

[Frieder, et al., 2000] O. Frieder, D. Grossman, A. Chowdhury, and G. Frieder, Efficiency Considerations in Very Large Information Retrieval Servers, Journal of Digital Information, (British Computer Society), 1(5).

[Goharian et al., 2000] N. Goharian, T. El-Ghazawi, D. Grossman, A. Chowdhury, On the Enhancements of a Sparse Matrix Information Retrieval Approach, PDPTA'2000.

[Goharian et al., 1999] N. Goharian, T. El-Ghazawi, D. Grossman, On the Implementation of Information Retrieval as Sparse Matrix Application, PDPTA'99.

[Goldman et al., 1998] R. Goldman, N. Shivakumar, S. Venkatasubramanian, H. Garcia-Molina, Proximity Search in Databases, VLDB'98, 26-37.

[Peters, 1991] Peters, Sparse matrix vector multiplication technique on the IBM 3090 VP, Parallel Computing 17.

[Petition et al., 1993] S. Petition, Y. Sood, K. Wu, W. Ferng, Basic sparse matrix computations on the CM-5, J. Mod. Phys., C(1).

[Stone, 1987] H. Stone, Parallel querying of large databases: A case study, IEEE Computer.

[UMich,1997] www.hti.umich.edu/relig/kjv/prox.html

[Witten et al., 1994] I. Witten, A. Moffat, and T. Bell, Managing Gigabytes. Van Nostrand Reinhold.