Manipulate Sparse Matrix Efficiently

Xiaohua (Davis) Zhou
College of Information Science & Technology, Drexel University
3141 Chestnut Street, Philadelphia, PA 19104, USA
xiaohua.zhou@drexel.edu

(Version 1.3, June 15, 2007)

1. Overview

Colt is a well-known Java-based package for sparse matrix. However, it does not fit text mining applications very well for two reasons. First, a frequently used operation in text mining applications is to get all terms and their frequency in a given document; colt is not efficient to execute this function. Second, colt uses hash method to implement sparse matrix and all data elements are kept in memory and thus not suitable for large text collections.

The dragon toolkit implements sparse matrix especially for text retrieval and mining use. Basically, it provides three types of sparse matrix, *flat sparse matrix*, s*uper sparse matrix*, and *giant sparse matrix*. The flat sparse matri*x* is very similar to the colt matrix. It loads all data into memory and runs very fast, but fit for small dataset only. The super sparse matrix and giant sparse matrix reside on disk-based files. Both have two files for a matrix. One is matrix file with the extension ".matrix" and the other is index file with the extension ".index". The index file is for fast loading purpose; its information is actually redundant and can be generated from the matrix file. The super sparse matrix loads index into the memory and caches given number of recent –read rows. The giant sparse matrix loads nothing into the memory except caching the last-read row. One can select appropriate sparse matrix for an applications based on the reading pattern, collection size, available physical memory as well as the expectation on reading speed.

With such an implementation of sparse matrix, the dragon toolkit is scalable to large-scale text retrieval and mining applications very well. It can handle hundred thousands of documents with very limited memory.

2. Installation

After installing the dragon toolkit, all functions of sparse matrix are available. If only sparse matrix is wanted, just download Dragon Toolkit Core Library (dragontool.jar) add the path of this jar file to the CLASSPATH environment variable.

3. Format of Matrix Files

In this section, I briefly introduce the format of matrix file and index file for both super sparse matrix and giant sparse matrix. If you are not interested in such details, skip this section and simply use provided API to access to the data stored in matrix files.

Both matrix file and index file are binary files and contain two sections, header and body. The header is the same for two files. It consists of twelve bytes which are interpreted as three integers denoting number of rows, number of columns, and number of non-zero cells, respectively.

The body of the matrix file is comprised of all rows. As shown in Figure 1, each row begins with the row ID (starting from zero) and the length of the row (i.e. number of non-zero data elements in the row) and is then followed by data points. The row ID, length, and column index always occupy four bytes, the bytes used for the data cell depends on the type of the data. For example, an integer uses four bytes and a double type data needs eight bytes. Thus, the actual bytes used for a row is variable. The column index of each data elements in a row is always in increasing order.

The body of index file consists of fixed length row information. Each row occupies sixteen bytes which are interpreted as an integer (the row ID), a long integer (the offset of the row in the matrix file), and an integer (the number of non-zero data points in the row). Using the index file, both super sparse matrix and giant sparse matrix can quickly locate a row in the matrix file. The information provided by the index file actually can be generated from the matrix file, but for fast loading purpose, we always generate the index file in advance. For a huge matrix, it takes some time to generate the index file dynamically.

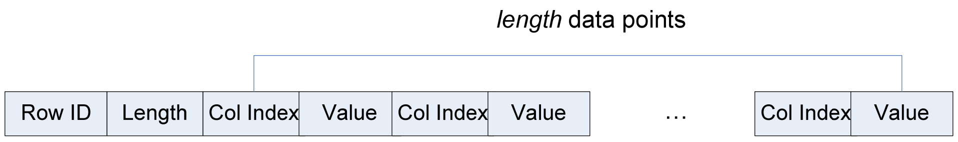

Figure 1: the illustration of the body of matrix file

4. Loading Matrix

If a matrix has been created and saved on a disk, it can be loaded through either programming API or XML-based configuration file. There are three types of matrix (flat, super and giant) and each type can be further classified according to the data type such as integer and double. In this section, all examples are assumed to load an integer-typed super sparse matrix.

4.1 Programming API

The loading of a matrix through programming API is quite straightforward. Some example code is shown in Figure 2. For super sparse matrix, it is also allowed to set the maximum number of rows to cache. Since it is slow to read a row from a binary file into memory, caching can make matrix-based applications run faster. The default cache size is 10,000 rows.

```
IntSuperSparseMatrix matrix;

//using both indexFile and matrixFile
matrix=new IntSuperSparseMatrix (indexFile, matrixFile);

// using only matrixFile
matrix=new IntSuperSparseMatrix(matrixFile) ;
matrix.setCache(5000) ;
```

Figure 2: The example of loading an integer-typed super sparse matrix

4.2 XML-based Configuration File

Regarding the background knowledge of configuraiton file, please refer to our short tutorial. The example shown in Figure 3 demostrate how to load an integer-typed super sparse matrix though XML-based configuration file. Basically two parameters, matrixpath and matrixkey, should be specified. The matrixpath parameter indicate the path of matrix files. It can be either relative path or absolute path. The matrixkey parameter infers the matrix file and the index file. In the example, the matrix key is “docterm” and thus the toolkit assumes the matrix file and index file are “docterm.matrix” and “docterm.index”, respectively.

```
<?xml version="1.0"?>
<configure>
    <intsupersparsematrix type="intsparsematrix" id="1">
        <param name="matrixpath" value="indexclustering/Newsgroup/tokenchar/all"/>
        <param name="matrixkey" value="docterm"/>
    </intsupersparsematrix>
</configure>
```

Figure 3: Example of configuration file loading an integer-typed super sparse matrix

5. Creating and Editing Matrix

The code in Figure 4 demonstrates the creation of sparse matrix. It is required to call the method *finalizeData* after adding new data points to the sparse matrix.

For the purpose efficiency and scalability, both super sparse matrix and giant sparse matrix temporarily save data points in a flat sparse matrix in memory. When the number of data points reaches certain threshold, it writes all data points the flat sparse matrix into disk files in batch mode. To control how often the batch writing should be executed, users can call the method *setFlushInterval* to set the threshold number of data points. The default threshold is 1,000,000. Users can also explicitly call the method *flush* to force the batch writing.

```
IntFlatSparseMatrix matrixA;
IntSuperSparseMatrix matrixB;
String matrixFile, indexFile;
Boolean mergeMode, miniMode ;

//create flat sparse matrix
matrixA=new  IntFlatSparseMatrix(mergeMode, miniMode) ;

//create super sparse matrix
matrixB=new IntSuperSparseMatrix(indexFile, matrixFile, mergeMode, miniMode)
matrixB.setFlushInterval(500000)

//add data to matrix
//add a cell at row 3 and column 4 to matrixA and matrixB
matrixA.add(3, 4, 2);
matrixB.add(3, 4, 2);
…

//finalize matrix
matrixA.finalizeData();
matrixA.close();
matrixB.finalizeData();
matrixB.close();
```

Figure 4: Example code of creating sparse matrix

All types of sparse matrix have two boolean parameters *mergeMode* and *miniMode* to control the behavior of matrix creation. The flat sparse matrix actually stores all data points in an array list. When a new data point is added, it needs to check if the cell (specified by row and column) exists. If not exists, simply add the data points in the end of the array list. Otherwise, merge the value of the existing data point. Certainly, it is not efficient to check the existence of a cell in a long array list. Thus, if it is for sure no duplicated cells will be added, please set this option to false. The option of *miniMode* is also related to the efficiency of the implementation. After data finalization, the flat sparse matrix creates an array each element of which indicates the starting position of the corresponding row in the array list. An extreme case is that the number of rows is very large (e.g., one million rows) while the number of data points is very small (e.g., one thousand). To create a huge array for a small number of data points is not worthy. Thus, the *miniMode* option is provided. In the aforementioned case, users can set *miniMode* option to false. Then the flat sparse matrix will not create the huge array after data finalization.

Flat sparse matrix is allowed to change the cell value after data finalization. However, super sparse matrix and giant sparse matrix are not allowed to do so directly. But users can add data points (existing or new) to existing super and giant sparse matrix.

6. Read Data Element

To read out the value of one random cell from a matrix is quite intuitive. For example, the method *getInt* and *getDouble* retrieve the integer value and double value from a specified cell, respectively. If the retrieved value is zero, it means the specified cell does not exist. However, for text retrieval and mining applications, a frequently used operation is to read out all terms and their frequency in a given document. The *getInt* and *getDouble* method are obviously not efficient to finish this task. Instead, we can use another method demonstrated in Figure 5.

The code in the Figure 5 is used to calculate the number of common terms in two documents. Because all data points in a sparse matrix are in an increasing column order, the time complexity of this task is O(n) rather than O(mn).

```
IntSparseMatrix matrix ;
int docA, docB ;
int arrFreqA[], arrFreqB[], arrColumnA[], arrColumnB[]
int a, b, count, termNum;

arrFreqA=matrix.getNonZeroIntScoresInRow(docA) ;
arrColumnA=matrix.getNonZeroColumnsInRow(docA) ;
arrFreqB=matrix.getNonZeroIntScoresInRow(docB) ;
arrColumnB=matrix.getNonZeroColumnsInRow(docB) ;

a=0;
b=0;
count=0
termNum=0;
while(a<arrFreqA.length && b<arrFreqB.length){
    if(arrColumnA[a]<arrColumnB[b])
        a++;
    else if(arrColumnA[a]<arrColumnB[b])
        b++;
     else{
        termNum++;
        count+=arrFreqA[a]+arrFreqB[b];
        a++;
        b++;
     }
}
```

Figure 5: Example code of reading sparse matrix

Sparse matrix is physically stored by rows. Thus, it is efficient to read out one row. However, sometimes, the operation has to be based on a column. In this case, users had better prepare the transposed matrix. The column operation on the current matrix is equal to the row operation on the transposed matrix.

7. Other Functions

The sparse matrix in the dragon toolkit also provides some common functions such as computing the cosine similarity of two rows. Please refer to the API documentations for all functionalities.