Bloom Filters

|  |  |
| --- | --- |
| **Exercise Dir** | ~/workspace/bloom\_filters |
| **Eclipse Proj** | bloom\_filters |
| **Java Files** | StockDividendDriver.java  Stock.java |
| **JAR File** | stringpairtest.jar |
| **Data File** | NYSE\_daily\_prices\_B.csv  NYSE\_daily\_prices\_C.csv |

**In this lab you will create a Bloom Filter, then apply the filter during the mapping of a MapReduce job. The results should leave you with a stock’s closing prices only on the datas that a dividend price was announced for that stock. Go ahead and open Eclipse before beginning.**

1.      Look over materials needed in the header above carefully.

1.1.   In Eclipse, open the bloom\_filter project. The project will show compiler errors, due to missing variables, etc.

1.2.   Open the file Stock.java. Notice the hashCode and equals methods are implemented so that two Stock objects are equal if they have the same stock symbol and date.

1.3.   Open the file StockDividendDriver.java and look at its run method. Notice this application consists of two MapReduce jobs: the first job will create the Bloom Filter and save it in a file named filters/dividendfilter. The second job will process all the stock prices and output only those stock prices whose symbol and date appear in the BloomFilter.

1.4.   Notice the BloomMapper and BloomReducer classes are fixmebed out for you and will represent the first MapReduce job that creates the BloomFilter file.

1.5.   Similarly, notice the StockFilterMapper and classes are fixmebed out and will represent the job that applies the filter to the stock prices.

2.      Define the BloomMapper

2.1.   Start by adding a field of type BloomFilter to the BloomMapper class:

private BloomFilter outputValue;

2.2.   In the setup method of BloomMapper, initialize the outputValue field with a new BloomFilter that has a vector size of 10000, uses 20 for the number of hash functions to consider, and uses the Murmur hash (declared in Hash.MURMUR\_HASH).

2.3.   The map method is written for you already. Notice that if the incoming stock symbol equals the stockSymbol field, then this stock needs to go into the BloomFilter. This is accomplished by adding a new Key object to the BloomFilter. That’s it for the map method. There are no <key/value> pairs to output yet.

2.4.   In the cleanup method of BloomMapper, output a single <key/value> pair that contains the BloomFilter as the value (notice the outputKey is a NullWritable instance):

context.write(outputKey, outputValue);

3.      Define the BloomReducer

3.1.   Add a BloomFilter field to BloomReducer named allValues:

private BloomFilter allValues;

3.2.   Initialize allValues in the setup method of BloomReducer. Use 10000 for the vector size, 20 for the number of hash functions, and Hash.MURMUR\_HASH as the hash function.

3.3.   The values coming in to the reduce method are BloomFilter instances. The BloomReducer needs to combine all of these BloomFilter instances into a single BloomFilter (i.e. the allValues field). Use the or method to combine each incoming BloomFilter with the allValues field:

for(BloomFilter filter : values) { allValues.or(filter);}

NOTE: The reduce method of BloomReducer does not output any <key/value> pairs. Notice in the run method the output format is set to NullOutputFormat.

3.4.   In the cleanup method, you need to output the BloomFilter to a file. To do this, start by creating the output file:

Configuration conf = context.getConfiguration();

Path path = new Path(FILTER\_FILE);

FSDataOutputStream out = path.getFileSystem(conf).create(path);

3.5.   Next, use the write method of BloomFilter to serialize the entire contents of

allValues to the file.

3.6.   Close the output stream.

3.7.   Save your changes the BloomReducer class is now ready to go!

4.      Test the BloomFilter Job

4.1.   Even though you are only halfway through the lab, you can test the first MapReduce job now. Build the project to create bloomfilter.jar.

4.2.   Put the “B” stocks into HDFS:

$ cd ~/materials/data

$ hdfs dfs -put NYSE\_daily\_prices\_\*.csv stocks

NOTE: If “File Already Exist” from the Introduction Lab, proceed to next step.

4.3.   Run the job (be sure to include a stock symbol as the first command line argument). For example:

$ yarn jar bloomfilter.jar BLU

4.4. You should see a new folder in HDFS named filters that contains a file named dividendfilter.

$ hdfs dfs -ls –R filters

Results:

Found 1 items

-rw-r--r-- 1 train hdfs1263 filters/dividendfilter

This is the BloomFilter output from the cleanup method of BloomReducer

5. Deserialize the BloomFilter

5.1. Notice that the setup method of StockFilterMapper has been started for you: the BloomFilter file has been retrieved from HDFS, and the stockSymbol field has been initialized.

5.2. Initialize the dividends field to a new BloomFilter with size 10000, 20 hash functions and the Hash.MURMUR\_HASH function.

5.3. Open filter\_file for input:

FileSystem fs = FileSystem.get(context.getConfiguration());

FSDataInputStream in = fs.open(filter\_file);

5.4. Use the readFields method of the dividends object to deserialize the contents of the file and input them into dividends.

5.5. Close the input stream.

6. Apply the BloomFilter to the Stocks

6.1. The map method of StockFilterMapper is only partially completed for you. Note the incoming data looks like:

NYSE,BGY,2010-02-08,10.25,10.39,9.94,10.28,600900,10.28

This line of text is split into an array of String objects.

6.2. If the stock symbol of the input data is equal to the stockSymbol field, then you need to check if this particular item appears in the BloomFilter. You need a Stock object to test this, so notice a Stock object is instantiated that contains the incoming symbol and date.

6.3. Instantiate a new BloomKey instance for the outputKey object.

6.4. Use the membershipTest method of BloomFilter to see if your new Key object is in dividends.

6.5. If the Key object is a member of dividends, then a match has been found and we want to output the closing price of this stock. Write out a <key/value> pair with the Stock object as the key, and the closing price as the value. (The closing price is at index 6 in your array of String objects.)

6.6. If the Stock object is not in dividends, then do not output anything.

6.7. Notice the StockFilterReducer is already written for you. Notice it checks for false positives before writing out any <key,value> pairs that it receives.

6.8. Save your changes to StockDividendDriver.java.

7. Run the Program

7.1. Build the project again to update bloomfilters.jar.

7.2. Run the job again using BLU as the stock symbol. (BLU just happens to have dividend values split across two input files.):

$ yarn jar bloomfilter.jar BLU

7.3. You should see a new folder in HDFS named bloomoutput:

$ hdfs dfs -ls –R bloomoutput;

7.4. View the output file part-r-00000. The tail of the file should look like:

BLU,2007-12-27,5.21

BLU,2008-04-09,4.76

BLU,2008-07-09,4.3

BLU,2008-10-08,2.65

BLU,2008-12-24,2.19

BLU,2009-04-07,2.22

Result: You have just written a MapReduce application that uses a BloomFilter to minimize the amount of data output from the Mapper. You could have performed this task using a single MapReduce job, but without a filter you would have to store a lot of data in memory in the Mapper, or pass a lot more information across the network and have the Reducer filter the output.

**END**