Program – 5

AIM: Write a program for bloom-filtering in Hadoop.

Introduction & Theory

Bloom Filter

A Bloom filter is a space-efficient probabilistic data structure, conceived by Burton Howard Bloom in 1970, that is used to test whether an element is a member of a set. False positive matches are possible, but false negatives are not – in other words, a query returns either "possibly in set" or "definitely not in set". Elements can be added to the set, but not removed (though this can be addressed with a "counting" filter); the more elements that are added to the set, the larger the probability of false positives

Unlike a standard hash table using open addressing for collision resolution, a Bloom filter of a fixed size can represent a set with an arbitrarily large number of elements; adding an element never fails due to the data structure "filling up". However, the false positive rate increases steadily as elements are added until all bits in the filter are set to 1, at which point all queries yield a positive result. With open addressing hashing, false positives are never produced, but performance steadily deteriorates until it approaches linear search.

Union and intersection of Bloom filters with the same size and set of hash functions can be implemented with bitwise OR and AND operations respectively. The union operation on Bloom filters is lossless in the sense that the resulting Bloom filter is the same as the Bloom filter created from scratch using the union of the two sets. The intersect operation satisfies a weaker property: the false positive probability in the resulting Bloom filter is at most the false-positive probability in one of the constituent Bloom filters, but may be larger than the false positive probability in the Bloom filter created from scratch using the intersection of the two sets. (interesting properties: source Wikipedia)

Below are the steps for MapReduce data flow:

- **Step 1:** One block is processed by one **mapper** at a time. In the mapper, a developer can specify his own business logic as per the requirements. In this manner, Map runs on all the nodes of the cluster and process the data blocks in parallel.
- **Step 2:** Output of Mapper also known as intermediate output is written to the local disk. An output of mapper is not stored on HDFS as this is temporary data and **writing on HDFS** will create unnecessary many copies.
- **Step 3:** Output of mapper is shuffled to **reducer** node (which is a normal slave node but reduce phase will run here hence called as reducer node). The shuffling/copying is a physical movement of data which is done over the network.
- **Step 4:** Once all the mappers are finished and their output is shuffled on reducer nodes then this intermediate output is merged & sorted. Which is then provided as input to reduce phase.
- **Step 5:** Reduce is the second phase of processing where the user can specify his own custom business logic as per the requirements. An input to a reducer is provided from all the mappers. An output of reducer is the final output, which is written on HDFS

Program – 5

Code

Bloom Filter Class

```java
package utils;

import java.io.Serializable;
import java.nio.charset.Charset;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.BitSet;
import java.util.Collection;


public class FilterBloom<E> implements Serializable {
    private BitSet bitset;
    private int bitSetSize;
    private double bitsPerElement;
    private int expectedNumberOfFilterElements; // expected (maximum)
number of elements to be added
    private int numberOfAddedElements; // number of elements actually
added to the Bloom filter
    private int k; // number of hash functions

    static final Charset charset = Charset.forName("UTF-8"); //
encoding used for storing hash values as strings

    static final String hashName = "MD5"; // MD5 gives good enough
accuracy in most circumstances. Change to SHA1 if it's needed
    static final MessageDigest digestFunction;
    static { // The digest method is reused between instances
        MessageDigest tmp;
        try {
            tmp = java.security.MessageDigest.getInstance(hashName);
        } catch (NoSuchAlgorithmException e) {
            tmp = null;
        }
        digestFunction = tmp;
    }

    /**
     * Constructs an empty Bloom filter. The total length of the
Bloom filter will be
     * c*n.
     *
     * @param c is the number of bits used per element.
     * @param n is the expected number of elements the filter will
contain.
     * @param k is the number of hash functions used.
     */
    public FilterBloom(double c, int n, int k) {
        this.expectedNumberOfFilterElements = n;
        this.k = k;
        this.bitsPerElement = c;
        this.bitSetSize = (int)Math.ceil(c * n);
        numberOfAddedElements = 0;
        this.bitset = new BitSet(bitSetSize);
```

Program – 5

```
    }

    /**
     * Constructs an empty Bloom filter. The optimal number of hash
functions (k) is estimated from the total size of the Bloom
     * and the number of expected elements.
     *
     * @param bitSetSize defines how many bits should be used in
total for the filter.
     * @param expectedNumberOElements defines the maximum number of
elements the filter is expected to contain.
     */
    public FilterBloom(int bitSetSize, int expectedNumberOElements) {
        this(bitSetSize / (double)expectedNumberOElements,
                expectedNumberOElements,
                (int) Math.round((bitSetSize /
(double)expectedNumberOElements) * Math.log(2.0)));
    }

    /**
     * Constructs an empty Bloom filter with a given false positive
probability. The number of bits per
     * element and the number of hash functions is estimated
     * to match the false positive probability.
     *
     * @param falsePositiveProbability is the desired false positive
probability.
     * @param expectedNumberOfElements is the expected number of
elements in the Bloom filter.
     */
    public FilterBloom(double falsePositiveProbability, int
expectedNumberOfElements) {
        this(Math.ceil(-(Math.log(falsePositiveProbability) /
Math.log(2))) / Math.log(2), // c = k / ln(2)
                expectedNumberOfElements,
                (int)Math.ceil(-(Math.log(falsePositiveProbability) /
Math.log(2)))); // k = ceil(-log_2(false prob.))
    }

    /**
     * Construct a new Bloom filter based on existing Bloom filter
data.
     *
     * @param bitSetSize defines how many bits should be used for the
filter.
     * @param expectedNumberOfFilterElements defines the maximum
number of elements the filter is expected to contain.
     * @param actualNumberOfFilterElements specifies how many
elements have been inserted into the <code>filterData</code> BitSet.
     * @param filterData a BitSet representing an existing Bloom
filter.
     */
    public FilterBloom(int bitSetSize, int
expectedNumberOfFilterElements, int actualNumberOfFilterElements,
BitSet filterData) {
        this(bitSetSize, expectedNumberOfFilterElements);
        this.bitset = filterData;
```

Program – 5

```
        this.numberOfAddedElements = actualNumberOfFilterElements;
    }

    /**
     * Generates a digest based on the contents of a String.
     *
     * @param val specifies the input data.
     * @param charset specifies the encoding of the input data.
     * @return digest as long.
     */
    public static int createHash(String val, Charset charset) {
        return createHash(val.getBytes(charset));
    }

    /**
     * Generates a digest based on the contents of a String.
     *
     * @param val specifies the input data. The encoding is expected
to be UTF-8.
     * @return digest as long.
     */
    public static int createHash(String val) {
        return createHash(val, charset);
    }

    /**
     * Generates a digest based on the contents of an array of bytes.
     *
     * @param data specifies input data.
     * @return digest as long.
     */
    public static int createHash(byte[] data) {
        return createHashes(data, 1)[0];
    }

    /**
     * Generates digests based on the contents of an array of bytes
and splits the result into 4-byte int's and store them in an array.
The
     * digest function is called until the required number of int's
are produced. For each call to digest a salt
     * is prepended to the data. The salt is increased by 1 for each
call.
     *
     * @param data specifies input data.
     * @param hashes number of hashes/int's to produce.
     * @return array of int-sized hashes
     */
    public static int[] createHashes(byte[] data, int hashes) {
        int[] result = new int[hashes];

        int k = 0;
        byte salt = 0;
        while (k < hashes) {
            byte[] digest;
            synchronized (digestFunction) {
                digestFunction.update(salt);
```

Program – 5

```
                salt++;
                digest = digestFunction.digest(data);
            }

            for (int i = 0; i < digest.length/4 && k < hashes; i++) {
                int h = 0;
                for (int j = (i*4); j < (i*4)+4; j++) {
                    h <<= 8;
                    h |= ((int) digest[j]) & 0xFF;
                }
                result[k] = h;
                k++;
            }
        }
        return result;
    }

    /**
     * Compares the contents of two instances to see if they are
equal.
     *
     * @param obj is the object to compare to.
     * @return True if the contents of the objects are equal.
     */
    @Override
    public boolean equals(Object obj) {
        if (obj == null) {
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
        final FilterBloom<E> other =  (FilterBloom<E>) obj;
        if (this.expectedNumberOfFilterElements !=
other.expectedNumberOfFilterElements) {
            return false;
        }
        if (this.k != other.k) {
            return false;
        }
        if (this.bitSetSize != other.bitSetSize) {
            return false;
        }
        if (this.bitset != other.bitset && (this.bitset == null ||
!this.bitset.equals(other.bitset))) {
            return false;
        }
        return true;
    }

    /**
     * Calculates a hash code for this class.
     * @return hash code representing the contents of an instance of
this class.
     */
    @Override
    public int hashCode() {
```

Program – 5

```
        int hash = 7;
        hash = 61 * hash + (this.bitset != null ?
this.bitset.hashCode() : 0);
        hash = 61 * hash + this.expectedNumberOfFilterElements;
        hash = 61 * hash + this.bitSetSize;
        hash = 61 * hash + this.k;
        return hash;
    }


    /**
     * Calculates the expected probability of false positives based
on
     * the number of expected filter elements and the size of the
Bloom filter.
     * <br /><br />
     * The value returned by this method is the <i>expected</i> rate
of false
     * positives, assuming the number of inserted elements equals the
number of
     * expected elements. If the number of elements in the Bloom
filter is less
     * than the expected value, the true probability of false
positives will be lower.
     *
     * @return expected probability of false positives.
     */
    public double expectedFalsePositiveProbability() {
        return
getFalsePositiveProbability(expectedNumberOfFilterElements);
    }

    /**
     * Calculate the probability of a false positive given the
specified
     * number of inserted elements.
     *
     * @param numberOfElements number of inserted elements.
     * @return probability of a false positive.
     */
    public double getFalsePositiveProbability(double
numberOfElements) {
        // (1 - e^(-k * n / m)) ^ k
        return Math.pow((1 - Math.exp(-k * (double) numberOfElements
                / (double) bitSetSize)), k);

    }

    /**
     * Get the current probability of a false positive. The
probability is calculated from
     * the size of the Bloom filter and the current number of
elements added to it.
     *
     * @return probability of false positives.
     */
    public double getFalsePositiveProbability() {
```

Program – 5

```
        return getFalsePositiveProbability(numberOfAddedElements);
    }


    /**
     * Returns the value chosen for K.<br />
     * <br />
     * K is the optimal number of hash functions based on the size
     * of the Bloom filter and the expected number of inserted
elements.
     *
     * @return optimal k.
     */
    public int getK() {
        return k;
    }

    /**
     * Sets all bits to false in the Bloom filter.
     */
    public void clear() {
        bitset.clear();
        numberOfAddedElements = 0;
    }

    /**
     * Adds an object to the Bloom filter. The output from the
object's
     * toString() method is used as input to the hash functions.
     *
     * @param element is an element to register in the Bloom filter.
     */
    public void add(E element) {
        add(element.toString().getBytes(charset));
    }

    /**
     * Adds an array of bytes to the Bloom filter.
     *
     * @param bytes array of bytes to add to the Bloom filter.
     */
    public void add(byte[] bytes) {
        int[] hashes = createHashes(bytes, k);
        for (int hash : hashes)
            bitset.set(Math.abs(hash % bitSetSize), true);
        numberOfAddedElements ++;
    }

    /**
     * Adds all elements from a Collection to the Bloom filter.
     * @param c Collection of elements.
     */
    public void addAll(Collection<? extends E> c) {
        for (E element : c)
            add(element);
    }

```

Program – 5

```
    /**
     * Returns true if the element could have been inserted into the
Bloom filter.
     * Use getFalsePositiveProbability() to calculate the probability
of this
     * being correct.
     *
     * @param element element to check.
     * @return true if the element could have been inserted into the
Bloom filter.
     */
    public boolean contains(E element) {
        return contains(element.toString().getBytes(charset));
    }

    /**
     * Returns true if the array of bytes could have been inserted
into the Bloom filter.
     * Use getFalsePositiveProbability() to calculate the probability
of this
     * being correct.
     *
     * @param bytes array of bytes to check.
     * @return true if the array could have been inserted into the
Bloom filter.
     */
    public boolean contains(byte[] bytes) {
        int[] hashes = createHashes(bytes, k);
        for (int hash : hashes) {
            if (!bitset.get(Math.abs(hash % bitSetSize))) {
                return false;
            }
        }
        return true;
    }

    /**
     * Returns true if all the elements of a Collection could have
been inserted
     * into the Bloom filter. Use getFalsePositiveProbability() to
calculate the
     * probability of this being correct.
     * @param c elements to check.
     * @return true if all the elements in c could have been inserted
into the Bloom filter.
     */
    public boolean containsAll(Collection<? extends E> c) {
        for (E element : c)
            if (!contains(element))
                return false;
        return true;
    }

    /**
     * Read a single bit from the Bloom filter.
     * @param bit the bit to read.
     * @return true if the bit is set, false if it is not.
```

Program – 5

```
     */
    public boolean getBit(int bit) {
        return bitset.get(bit);
    }

    /**
     * Set a single bit in the Bloom filter.
     * @param bit is the bit to set.
     * @param value If true, the bit is set. If false, the bit is
cleared.
     */
    public void setBit(int bit, boolean value) {
        bitset.set(bit, value);
    }

    /**
     * Return the bit set used to store the Bloom filter.
     * @return bit set representing the Bloom filter.
     */
    public BitSet getBitSet() {
        return bitset;
    }

    /**
     * Returns the number of bits in the Bloom filter. Use count() to
retrieve
     * the number of inserted elements.
     *
     * @return the size of the bitset used by the Bloom filter.
     */
    public int size() {
        return this.bitSetSize;
    }

    /**
     * Returns the number of elements added to the Bloom filter after
it
     * was constructed or after clear() was called.
     *
     * @return number of elements added to the Bloom filter.
     */
    public int count() {
        return this.numberOfAddedElements;
    }

    /**
     * Returns the expected number of elements to be inserted into
the filter.
     * This value is the same value as the one passed to the
constructor.
     *
     * @return expected number of elements.
     */
    public int getExpectedNumberOfElements() {
        return expectedNumberOfFilterElements;
    }

```

Program – 5

```
    /**
     * Get expected number of bits per element when the Bloom filter
is full. This value is set by the constructor
     * when the Bloom filter is created. See also
getBitsPerElement().
     *
     * @return expected number of bits per element.
     */
    public double getExpectedBitsPerElement() {
        return this.bitsPerElement;
    }

    /**
     * Get actual number of bits per element based on the number of
elements that have currently been inserted and the length
     * of the Bloom filter. See also getExpectedBitsPerElement().
     *
     * @return number of bits per element.
     */
    public double getBitsPerElement() {
        return this.bitSetSize / (double)numberOfAddedElements;
    }
}
```

Mapper

```
package bfcode;
import java.lang.System.*;
import java.io.IOException;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.Mapper;
import utils.*;
public class FilterMapper extends Mapper<LongWritable, Text, Text,
NullWritable> {
    FilterBloom<String> filter;
    @Override
    protected void setup(org.apache.hadoop.mapreduce.Mapper.Context
context)
            throws IOException, InterruptedException {
        super.setup(context);
        double falsePositiveProbability = 0.1;
        int expectedNumberOfElements = 100;
        filter = new FilterBloom<String>(falsePositiveProbability,
expectedNumberOfElements);
        filter.add("bad service");
        filter.add("iron man");
        filter.add("marvel");
        filter.add("end game");
    }
    protected void map(LongWritable key, Text value, Context context)
            throws java.io.IOException, InterruptedException {

        String[] tokens = value.toString().split(",");
        for(String token :tokens){
```

Program – 5

```java
                System.out.println(token);
                if(filter.contains(token)){
                    context.write(value, NullWritable.get());
                }
            }
        }
}
```

Main

```java
package bfcode;
import java.io.IOException;
import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.IntWritable;
import org.apache.hadoop.io.NullWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
import org.apache.hadoop.mapreduce.Counter;
import org.apache.hadoop.mapreduce.CounterGroup;
import org.apache.hadoop.mapreduce.Counters;
import org.apache.hadoop.mapreduce.Job;

public class BloomFilter {

    public static void main(String[] args)
            throws IOException, ClassNotFoundException,
InterruptedException {
        if (args.length != 2) {
            System.err.println("Usage: FilterJob <input path> <output path>");
            System.exit(-1);
        }

        Job job = new Job();
        job.setJarByClass(bfcode.FilterMapper.class);
        job.setJobName("Customer Complaint Filter");

        FileInputFormat.addInputPath(job, new Path(args[0]));
        FileOutputFormat.setOutputPath(job, new Path(args[1]));

        job.setMapperClass(bfcode.FilterMapper.class);
        job.setOutputKeyClass(Text.class);
        job.setOutputValueClass(NullWritable.class);
        job.waitForCompletion(true);
    }
}
```

Program – 5

Output

data.txt

wordc.txt | stopwords.txt | data.txt

```
10/11/2012,r..@yahoo.com,3,I was really satisfied with the customer
service at the store
09/12/2012,x..@yahoo.com,3,Basic needs met at the store
10/11/2012,r.@gmail.com,4,Excellent
09/12/2012,x..@hotmail.com,5,Great Collection
01/02/2013,dan...@xyz.com,1,bad service
09/12/2012,zz..@hotmail.com,5,bad service
01/02/2013,d...@gmail.com,4,Good collection and service at the
store
```

hduser@rinzler-jarvis: ~/CodeLib/BloomFilter/src

```
hduser@rinzler-jarvis:~/CodeLib/BloomFilter/src$ hadoop jar bloom.jar bfcode.Blo
omFilter input/marvel.txt output/blooming
10/11/2012
r..@yahoo.com
3
I was really satisfied with the movie
 marvel doesnt let you down.
09/12/2012
x..@yahoo.com
3
Basic needs met at the store
10/11/2012
r.@gmail.com
4
Excellent
09/12/2012
x..@hotmail.com
5
Great Collection
 thor was amazing
01/02/2013
dan...@xyz.com
1
bad service
09/12/2012
zz..@hotmail.com
5
bad service
01/02/2013
d...@gmail.com
4
cant wait for end game

hduser@rinzler-jarvis:~/CodeLib/BloomFilter/src$ hdfs dfs -cat output/blooming/p
art*
01/02/2013,dan...@xyz.com,1,bad service
09/12/2012,zz..@hotmail.com,5,bad service
hduser@rinzler-jarvis:~/CodeLib/BloomFilter/src$ hdfs dfs -copyToLocal output/bl
ooming ~/output/blooming
hduser@rinzler-jarvis:~/CodeLib/BloomFilter/src$
```

Findings and Learnings:

1. We learned how map-reduce works.
2. We learned how to code using hadoop in Java.
3. We have learned the working of bloom filters
4. We have successfully implemented bloom filters