

## Bloom Filter Class using MD5 as Hash Function

```
1: package com.skjegstad.utils;
2:
3: import java.io.Serializable;
4: import java.nio.charset.Charset;
5: import java.security.MessageDigest;
6: import java.security.NoSuchAlgorithmException;
7: import java.util.BitSet;
8: import java.util.Collection;
9:
10: // @author Magnus Skjegstad <magnus@skjegstad.com>
11: public class BloomFilter<E> implements Serializable {
12:     private BitSet bitset;
13:     private int bitSetSize;
14:     private double bitsPerElement;
15:     private int expectedNumberOfFilterElements; // expected (maximum) number of
        elements to be added
16:     private int numberOfAddedElements; // number of elements actually added to
        the Bloom filter
17:     private int k; // number of hash functions
18:
19:     static final Charset charset = Charset.forName("UTF-8"); // encoding used
        for storing hash values as strings
20:
21:     static final String hashName = "MD5"; // MD5 gives good enough accuracy in
        most circumstances. Change to SHA1 if it's needed
22:     static final MessageDigest digestFunction;
23:     static { // The digest method is reused between instances
24:         MessageDigest tmp;
25:         try {
26:             tmp = java.security.MessageDigest.getInstance(hashName);
27:         } catch (NoSuchAlgorithmException e) {
28:             tmp = null;
29:         }
30:         digestFunction = tmp;
31:     }
32:
33:     /**
34:      * Constructs an empty Bloom filter. The total length of the Bloom filter
        will be
35:      * c*n.
36:      *
37:      * @param c is the number of bits used per element.
38:      * @param n is the expected number of elements the filter will contain.
39:      * @param k is the number of hash functions used.
40:      */
41:     public BloomFilter(double c, int n, int k) {
42:         this.expectedNumberOfFilterElements = n;
43:         this.k = k;
44:         this.bitsPerElement = c;
45:         this.bitSetSize = (int) Math.ceil(c * n);
46:         numberOfAddedElements = 0;
47:         this.bitset = new BitSet(bitSetSize);
48:     }
49:
50:     /**
```

```

51:      * Constructs an empty Bloom filter. The optimal number of hash functions
      (k) is estimated from the total size of the Bloom
52:      * and the number of expected elements.
53:      *
54:      * @param bitSetSize defines how many bits should be used in total for the
      filter.
55:      * @param expectedNumberOfElements defines the maximum number of elements the
      filter is expected to contain.
56:      */
57:      public BloomFilter(int bitSetSize, int expectedNumberOfElements) {
58:          this(bitSetSize / (double)expectedNumberOfElements,
59:              expectedNumberOfElements,
60:              (int) Math.round((bitSetSize / (double)expectedNumberOfElements) *
      Math.log(2.0)));
61:      }
62:
63:      /**
64:       * Constructs an empty Bloom filter with a given false positive probability.
      The number of bits per
65:       * element and the number of hash functions is estimated
66:       * to match the false positive probability.
67:       *
68:       * @param falsePositiveProbability is the desired false positive
      probability.
69:       * @param expectedNumberOfElements is the expected number of elements in the
      Bloom filter.
70:       */
71:      public BloomFilter(double falsePositiveProbability, int
      expectedNumberOfElements) {
72:          this(Math.ceil(-(Math.log(falsePositiveProbability) / Math.log(2))) /
      Math.log(2), // c = k / ln(2)
73:              expectedNumberOfElements,
74:              (int) Math.ceil(-(Math.log(falsePositiveProbability) /
      Math.log(2))))); // k = ceil(-log2(false prob.))
75:      }
76:
77:      /**
78:       * Construct a new Bloom filter based on existing Bloom filter data.
79:       *
80:       * @param bitSetSize defines how many bits should be used for the filter.
81:       * @param expectedNumberOfFilterElements defines the maximum number of
      elements the filter is expected to contain.
82:       * @param actualNumberOfFilterElements specifies how many elements have been
      inserted into the <code>filterData</code> BitSet.
83:       * @param filterData a BitSet representing an existing Bloom filter.
84:       */
85:      public BloomFilter(int bitSetSize, int expectedNumberOfFilterElements, int
      actualNumberOfFilterElements, BitSet filterData) {
86:          this(bitSetSize, expectedNumberOfFilterElements);
87:          this.bitset = filterData;
88:          this.numberOfAddedElements = actualNumberOfFilterElements;
89:      }
90:
91:      /**
92:       * Generates a digest based on the contents of a String.
93:       *
94:       * @param val specifies the input data.

```

```

95:      * @param charset specifies the encoding of the input data.
96:      * @return digest as long.
97:      */
98:      public static int createHash(String val, Charset charset) {
99:          return createHash(val.getBytes(charset));
100:      }
101:
102:      /**
103:       * Generates a digest based on the contents of a String.
104:       *
105:       * @param val specifies the input data. The encoding is expected to be
UTF-8.
106:       * @return digest as long.
107:       */
108:      public static int createHash(String val) {
109:          return createHash(val, charset);
110:      }
111:
112:      /**
113:       * Generates a digest based on the contents of an array of bytes.
114:       *
115:       * @param data specifies input data.
116:       * @return digest as long.
117:       */
118:      public static int createHash(byte[] data) {
119:          return createHashes(data, 1)[0];
120:      }
121:
122:      /**
123:       * 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
124:       * digest function is called until the required number of int's are
produced. For each call to digest a salt
125:       * is prepended to the data. The salt is increased by 1 for each call.
126:       *
127:       * @param data specifies input data.
128:       * @param hashes number of hashes/int's to produce.
129:       * @return array of int-sized hashes
130:       */
131:      public static int[] createHashes(byte[] data, int hashes) {
132:          int[] result = new int[hashes];
133:
134:          int k = 0;
135:          byte salt = 0;
136:          while (k < hashes) {
137:              byte[] digest;
138:              synchronized (digestFunction) {
139:                  digestFunction.update(salt);
140:                  salt++;
141:                  digest = digestFunction.digest(data);
142:              }
143:
144:              for (int i = 0; i < digest.length/4 && k < hashes; i++) {
145:                  int h = 0;
146:                  for (int j = (i*4); j < (i*4)+4; j++) {
147:                      h <= 8;
148:                      h |= ((int) digest[j]) & 0xFF;

```

```

149:                }
150:                result[k] = h;
151:                k++;
152:            }
153:        }
154:        return result;
155:    }
156:
157:    /**
158:     * Compares the contents of two instances to see if they are equal.
159:     *
160:     * @param obj is the object to compare to.
161:     * @return True if the contents of the objects are equal.
162:     */
163:    @Override
164:    public boolean equals(Object obj) {
165:        if (obj == null) {
166:            return false;
167:        }
168:        if (getClass() != obj.getClass()) {
169:            return false;
170:        }
171:        final BloomFilter<E> other = (BloomFilter<E>) obj;
172:        if (this.expectedNumberOfFilterElements !=
173:            other.expectedNumberOfFilterElements) {
174:            return false;
175:        }
176:        if (this.k != other.k) {
177:            return false;
178:        }
179:        if (this.bitSetSize != other.bitSetSize) {
180:            return false;
181:        }
182:        if (this.bitset != other.bitset && (this.bitset == null ||
183:            !this.bitset.equals(other.bitset))) {
184:            return false;
185:        }
186:        return true;
187:    }
188:
189:    /**
190:     * Calculates a hash code for this class.
191:     * @return hash code representing the contents of an instance of this
192:     class.
193:     */
194:    @Override
195:    public int hashCode() {
196:        int hash = 7;
197:        hash = 61 * hash + (this.bitset != null ? this.bitset.hashCode() :
198:            0);
199:        hash = 61 * hash + this.expectedNumberOfFilterElements;
200:        hash = 61 * hash + this.bitSetSize;
201:        hash = 61 * hash + this.k;
202:        return hash;
203:    }

```

```

202:      /**
203:       * Calculates the expected probability of false positives based on
204:       * the number of expected filter elements and the size of the Bloom
        filter.
205:       * <br /><br />
206:       * The value returned by this method is the <i>expected</i> rate of
        false
207:       * positives, assuming the number of inserted elements equals the
        number of
208:       * expected elements. If the number of elements in the Bloom filter is
        less
209:       * than the expected value, the true probability of false positives
        will be lower.
210:       *
211:       * @return expected probability of false positives.
212:       */
213:     public double expectedFalsePositiveProbability() {
214:         return getFalsePositiveProbability(expectedNumberOfFilterElements);
215:     }
216:
217:     /**
218:      * Calculate the probability of a false positive given the specified
219:      * number of inserted elements.
220:      *
221:      * @param numberOfElements number of inserted elements.
222:      * @return probability of a false positive.
223:      */
224:     public double getFalsePositiveProbability(double numberOfElements) {
225:         // (1 - e^(-k * n / m)) ^ k
226:         return Math.pow((1 - Math.exp(-k * (double) numberOfElements
227:             / (double) bitSetSize)), k);
228:     }
229:
230:
231:     /**
232:      * Get the current probability of a false positive. The probability is
        calculated from
233:      * the size of the Bloom filter and the current number of elements
        added to it.
234:      *
235:      * @return probability of false positives.
236:      */
237:     public double getFalsePositiveProbability() {
238:         return getFalsePositiveProbability(numberOfAddedElements);
239:     }
240:
241:
242:     /**
243:      * Returns the value chosen for K.<br />
244:      * <br />
245:      * K is the optimal number of hash functions based on the size
246:      * of the Bloom filter and the expected number of inserted elements.
247:      *
248:      * @return optimal k.
249:      */
250:     public int getK() {
251:         return k;

```

```

252:     }
253:
254:     /**
255:      * Sets all bits to false in the Bloom filter.
256:      */
257:     public void clear() {
258:         bitset.clear();
259:         numberOfAddedElements = 0;
260:     }
261:
262:     /**
263:      * Adds an object to the Bloom filter. The output from the object's
264:      * toString() method is used as input to the hash functions.
265:      *
266:      * @param element is an element to register in the Bloom filter.
267:      */
268:     public void add(E element) {
269:         add(element.toString().getBytes(charset));
270:     }
271:
272:     /**
273:      * Adds an array of bytes to the Bloom filter.
274:      *
275:      * @param bytes array of bytes to add to the Bloom filter.
276:      */
277:     public void add(byte[] bytes) {
278:         int[] hashes = createHashes(bytes, k);
279:         for (int hash : hashes)
280:             bitset.set(Math.abs(hash % bitSetSize), true);
281:         numberOfAddedElements++;
282:     }
283:
284:     /**
285:      * Adds all elements from a Collection to the Bloom filter.
286:      * @param c Collection of elements.
287:      */
288:     public void addAll(Collection<? extends E> c) {
289:         for (E element : c)
290:             add(element);
291:     }
292:
293:     /**
294:      * Returns true if the element could have been inserted into the Bloom
295:      * filter.
296:      *
297:      * Use getFalsePositiveProbability() to calculate the probability of
298:      * this
299:      * being correct.
300:      *
301:      * @param element element to check.
302:      * @return true if the element could have been inserted into the Bloom
303:      * filter.
304:      */
305:     public boolean contains(E element) {
306:         return contains(element.toString().getBytes(charset));
307:     }

```

```

306:      * Returns true if the array of bytes could have been inserted into the
      Bloom filter.
307:      * Use getFalsePositiveProbability() to calculate the probability of
      this
308:      * being correct.
309:      *
310:      * @param bytes array of bytes to check.
311:      * @return true if the array could have been inserted into the Bloom
      filter.
312:      */
313:      public boolean contains(byte[] bytes) {
314:          int[] hashes = createHashes(bytes, k);
315:          for (int hash : hashes) {
316:              if (!bitset.get(Math.abs(hash % bitSetSize))) {
317:                  return false;
318:              }
319:          }
320:          return true;
321:      }
322:
323:      /**
324:       * Returns true if all the elements of a Collection could have been
      inserted
325:       * into the Bloom filter. Use getFalsePositiveProbability() to
      calculate the
326:       * probability of this being correct.
327:       * @param c elements to check.
328:       * @return true if all the elements in c could have been inserted into
      the Bloom filter.
329:       */
330:      public boolean containsAll(Collection<? extends E> c) {
331:          for (E element : c)
332:              if (!contains(element))
333:                  return false;
334:          return true;
335:      }
336:
337:      /**
338:       * Read a single bit from the Bloom filter.
339:       * @param bit the bit to read.
340:       * @return true if the bit is set, false if it is not.
341:       */
342:      public boolean getBit(int bit) {
343:          return bitset.get(bit);
344:      }
345:
346:      /**
347:       * Set a single bit in the Bloom filter.
348:       * @param bit is the bit to set.
349:       * @param value If true, the bit is set. If false, the bit is cleared.
350:       */
351:      public void setBit(int bit, boolean value) {
352:          bitset.set(bit, value);
353:      }
354:
355:      /**
356:       * Return the bit set used to store the Bloom filter.

```

```

357:      * @return bit set representing the Bloom filter.
358:      */
359:      public BitSet getBitSet() {
360:          return bitset;
361:      }
362:
363:      /**
364:       * Returns the number of bits in the Bloom filter. Use count() to
retrieve
365:       * the number of inserted elements.
366:       *
367:       * @return the size of the bitset used by the Bloom filter.
368:       */
369:      public int size() {
370:          return this.bitSetSize;
371:      }
372:
373:      /**
374:       * Returns the number of elements added to the Bloom filter after it
375:       * was constructed or after clear() was called.
376:       *
377:       * @return number of elements added to the Bloom filter.
378:       */
379:      public int count() {
380:          return this.numberOfAddedElements;
381:      }
382:
383:      /**
384:       * Returns the expected number of elements to be inserted into the
filter.
385:       * This value is the same value as the one passed to the constructor.
386:       *
387:       * @return expected number of elements.
388:       */
389:      public int getExpectedNumberOfElements() {
390:          return expectedNumberOfFilterElements;
391:      }
392:
393:      /**
394:       * Get expected number of bits per element when the Bloom filter is
full. This value is set by the constructor
395:       * when the Bloom filter is created. See also getBitsPerElement().
396:       *
397:       * @return expected number of bits per element.
398:       */
399:      public double getExpectedBitsPerElement() {
400:          return this.bitsPerElement;
401:      }
402:
403:      /**
404:       * Get actual number of bits per element based on the number of
elements that have currently been inserted and the length
405:       * of the Bloom filter. See also getExpectedBitsPerElement().
406:       *
407:       * @return number of bits per element.
408:       */
409:      public double getBitsPerElement() {

```



```
410:         return this.bitSetSize / (double)numberOfAddedElements;
411:     }
412: }
```

## How to Use Bloom Filter Class

```
1: package com.skjegstad.utils;
2:
3: import java.util.ArrayList;
4: import java.util.List;
5: import java.util.Random;
6:
7: // @author Magnus Skjegstad
8:
9: public class BloomfilterBenchmark {
10:     static int elementCount = 50000; // Number of elements to test
11:
12:     public static void printStat(long start, long end) {
13:         double diff = (end - start) / 1000.0;
14:         System.out.println(diff + "s, " + (elementCount / diff) + "
elements/s");
15:     }
16:
17:     public static void main(String[] argv) {
18:
19:
20:         final Random r = new Random();
21:
22:         // Generate elements first
23:         List<String> existingElements = new ArrayList(elementCount);
24:         for (int i = 0; i < elementCount; i++) {
25:             byte[] b = new byte[200];
26:             r.nextBytes(b);
27:             existingElements.add(new String(b));
28:         }
29:
30:         List<String> nonExistingElements = new ArrayList(elementCount);
31:         for (int i = 0; i < elementCount; i++) {
32:             byte[] b = new byte[200];
33:             r.nextBytes(b);
34:             nonExistingElements.add(new String(b));
35:         }
36:
37:         BloomFilter<String> bf = new BloomFilter<String>(0.001, elementCount);
38:
39:         System.out.println("Testing " + elementCount + " elements");
40:         System.out.println("k is " + bf.getK());
41:
42:         // Add elements
43:         System.out.print("add(): ");
44:         long start_add = System.currentTimeMillis();
45:         for (int i = 0; i < elementCount; i++) {
46:             bf.add(existingElements.get(i));
47:         }
48:         long end_add = System.currentTimeMillis();
49:         printStat(start_add, end_add);
50:
51:         // Check for existing elements with contains()
52:         System.out.print("contains(), existing: ");
53:         long start_contains = System.currentTimeMillis();
54:         for (int i = 0; i < elementCount; i++) {
```

```

55:         bf.contains(existingElements.get(i));
56:     }
57:     long end_contains = System.currentTimeMillis();
58:     printStat(start_contains, end_contains);
59:
60:     // Check for existing elements with containsAll()
61:     System.out.print("containsAll(), existing: ");
62:     long start_containsAll = System.currentTimeMillis();
63:     for (int i = 0; i < elementCount; i++) {
64:         bf.contains(existingElements.get(i));
65:     }
66:     long end_containsAll = System.currentTimeMillis();
67:     printStat(start_containsAll, end_containsAll);
68:
69:     // Check for nonexisting elements with contains()
70:     System.out.print("contains(), nonexisting: ");
71:     long start_ncontains = System.currentTimeMillis();
72:     for (int i = 0; i < elementCount; i++) {
73:         bf.contains(nonExistingElements.get(i));
74:     }
75:     long end_ncontains = System.currentTimeMillis();
76:     printStat(start_ncontains, end_ncontains);
77:
78:     // Check for nonexisting elements with containsAll()
79:     System.out.print("containsAll(), nonexisting: ");
80:     long start_ncontainsAll = System.currentTimeMillis();
81:     for (int i = 0; i < elementCount; i++) {
82:         bf.contains(nonExistingElements.get(i));
83:     }
84:     long end_ncontainsAll = System.currentTimeMillis();
85:     printStat(start_ncontainsAll, end_ncontainsAll);
86:
87: }
88: }

```