Bloom Filter version 1.0 Component Specification

1.Design

A Bloom filter is a probabilistic data structure that can be used to test for set membership in constant space and constant time. It may return false positives, but never false negatives.

The Bloom Filter Component provides an implementation of this data structure. This Component is designed to be very similar to the Java Collections Framework, so that users will instantly get a feel for working with the API provided by this Component.

Operations supported by the Bloom Filter data structure are:

- addition of a new item
- test for membership of an item
- union of two Bloom filters
- intersection of two Bloom filters

This component doesn't implement *java.util.Set* interface, because it is not really a collection of elements, and a lot of operations, such as iterating through collection elements have no sense for Bloom filter.

Bloom Filter Creation
It is possible to create a Bloom filter either by specifying a capacity (maximum number of items to be inserted) and a maximum error rate, or by specifying a size for the bit vector and a number of hashing functions. When a Bloom filter is created by specifying a capacity and an error rate, the size of the bit vector and the number of hashing functions are computed to minimize the size of the bit vector while preserving the maximum error rate at the given capacity. Note that more items than the capacity may be inserted, but beyond the capacity, the guaranteed error rate no longer holds.

Hash Functions
The Bloom filter uses a family of hashing functions, represented by *HashFunctionFamily* interface. The Component provides the default implementation of this interface which is called *DefaultHashFunctionFamily*. It uses one algorithm seeded with *k* different integers to generate *k* different hash functions.
The family of hashing functions is pluggable.

Serialization
It is possible to convert a Bloom filter to and from a string representation that would be suitable for use in a plain text or HTML document. This string representation will contain only printable characters.

1.1Design Patterns

The Strategy Pattern. The BloomFilter class uses the hash function family class to calculate hash values. HashFunctionFamily interface and DefaultHashFunctionFamily class also participate in this pattern.

1.2Industry Standards

None

1.3Required Algorithms

Detailed descriptions of the Bloom filter algorithms can be found at http://en.wikipedia.org/wiki/Bloom_filter, http://www.cs.wisc.edu/~cao/papers/summary-cache/node8.html, and http://portal.acm.org/citation.cfm?id=362692&dl=ACM&coll=portal.

An **empty Bloom filter** is a bit array of *m* bits, all set to 0. There must also be *k* different hash functions defined, each of which maps a key value to one of the *m* array positions.

For a good hash function with a wide output, there should be little if any correlation between different bit-fields of such a hash, so this type of hash can be used to generate multiple "different" hash functions by slicing its output into multiple bit fields.

To **add** an element, feed it to each of the *k* hash functions to get *k* array positions. Set the bits at all these positions to 1.

To **query** for an element (test whether it is in the set), feed it to each of the *k* hash functions to get *k* array positions. If any of the bits at these positions are 0, the element is not in the set – if it were, then all the bits would have been set to 1 when it was inserted. If all are 1, then either the element is in the set, or the bits have been set to 1 during the insertion of other elements.

Unfortunately, **removing** an element from this simple Bloom filter is impossible. The element maps to *k* bits, and although setting any one of these *k* bits to zero suffices to remove it, this has the side effect of removing any other elements that map onto that bit, and we have no way of determining whether any such elements have been added. The result is a possibility of false negatives, which are not allowed.

Union and **intersection** of Bloom filters with the same size and set of hash functions can be implemented by performing bitwise OR and AND operations on the bit sets, respectively

1.3.1 Adding an item to the Bloom filter

```
int maxHash = getBitSetLength() - 1;

// compute the hash for each function in the family and set that bit.
for (int i = 0; i < hashFunctions.getFunctionCount(); i++) {
      bitSet.set(hashFunctions.computeHash(i, maxHash, item));
}
```

1.3.2 Checking if the Bloom filter contains the specified item

```
int maxHash = getBitSetLength() - 1;

// For each of the hash functions, compute the hash value and check if that bit
// is set.  If any of those bits is not set, the element is not contained, but
// if they're all set, the element is contained (or might be a false positive).
for (int i = 0;  i < hashFunctions.getFunctionCount(); ++i) {
int hash = hashFunctions.computeHash(i, maxHash, item);
     if (!bitSet.get(hash)) {
         return false;
     }
}

return true;
```

1.3.3 Building string representation of the Bloom filter's bit set

```
// Make string representation of bit set length
String lengthString = Integer.toString(this.getBitSetLength());

StringBuffer buffer = new StringBuffer(lengthString + ":");
// Loop through the bit set
for (int i = 0; i < this.bitSet.length();) {
      int sum = 0;
      int pow = 1;
      for (int j = 0; j < 6 && i < this.bitSet.length(); j++, i++) {
                  if (this.bitSet.get(i))
                              sum += pow;
                  pow <<= 1;
      }
      buffer.append(BASE64_CHARS.charAt(sum));
}
// Return result
return buffer.toString();
```

1.1.3.4 Restoring the Bloom filter's bit set from the string representation

```
// Split the string in 2 (there could be more than 1 ':' because it is a valid base 64 character).
String parts[] = bitSetString.split(":", 2);

// the first part is the length. The regular expression already checked that it is a number.
bitSetSize = Integer.parseInt(parts[0]);

// loop through all the base 64 characters and set the bitset
for (int i = 0, bitPos = 0; i < parts[1].length(); i++) {

    // get the decimal value of the current base 64 character.
    int sum = BASE64_CHARS.indexOf(parts[1].charAt(i));

    // convert the decimal value to binary and set the bits in bitset, finishing the loop if
    // it arrived to the bit set size.
    for (int j = 0; (j < 6) && (bitPos < bitSetSize); bitPos++, j++) {
        if ((sum & 1) == 1) {
            bitSet.set(bitPos);
        }
        sum >>= 1;
    }
}
```

1.4Component Class Overview

Class BloomFilter

This class represents an implementation of a Bloom filter.
A Bloom filter is a probabilistic data structure that can be used to test for set membership in constant space and constant time. It may return false positives, but never false negatives.

Interface HashFunctionFamily

This interface represents a family of hash functions used by BloomFilter.
The implementations of this interface should provide distinct hash functions. The algorithm of hashing may be the same but with different parameters. Hash functions should return hash for given Object, in the range from 0 to specified maximal value.

Class DefaultHashFunctionFamily

This class is a simple implementation of HashFunctionFamily interface.
Hash values calculated by it are based on the hash values returned by Object.hashCode() method. See computeHash() documentation for hash computing algorithm description. Implementations of hashCode() method of basic types are described in J2SDK javadocs. So there should be no problems while using this hash function family in any foreign languages(e.g JavaScript, PHP).

1.5Component Exception Definitions

Exception BloomFilterException

This exception is the base exception for all exceptions in the component.
It is never directly thrown.

Exception IncomatibleBloomFiltersException

This exception is thrown by the BloomFilter class when union or intersection operation is performed on two Bloom filters that are incompatible.
Two Bloom filters are meant to be incomptible if they do not have equal bit set lengths and hash function families.
This exception derives from BloomFilterException.

Exception BloomFilterSerializeException

This exception is thrown by BloomFilter and DefaultHashFunctionFamily when the object can’t be built from a serialized string because it has invalid data or wrong format.
This exception derives from BloomFilterException.

1.6Thread Safety

This component is not thread-safe. *BloomFilter* class is mutable because the bit set can be changed by its methods. External synchronization is needed to use this class with multiple threads.

DefaultHashFunctionFamily is immutable and so thread safe. All the Implementations of HashFunction Family must be immutable.

2.Environment Requirements

2.1Environment

- Windows 95/98/NT/2000/XP/2003
- Solaris 7
- RedHat Linux 7.1
- Java 1.4

2.2TopCoder Software Components

None used.

2.3Third Party Components

None used

3.Installation and Configuration

3.1Package Names

com.topcoder.bloom

3.2Configuration Parameters

None required

3.3Dependencies Configuration

None required

4.Usage Notes

4.1Required steps to test the component

- Extract the component distribution.
- Follow Dependencies Configuration.
- Execute ‘ant test’ within the directory that the distribution was extracted to.

4.2Required steps to use the component

None, see demo section

4.3Demo

```
// Create a Bloom filter with the specified capacity and maximal error rate
// and custom hash function family
BloomFilter filter00 = new BloomFilter(1000, 0.01f, DefaultHashFunctionFamily.class);

// Create a Bloom filter with the specified capacity and maximal error rate
BloomFilter filter1 = new BloomFilter(1000, 0.01f);
```

```
// Create a Bloom filter with the specified length of bit set - 10000
// and number of hash functions - 10
BloomFilter filter0 = new BloomFilter(10000, new DefaultHashFunctionFamily(10));


// Add some items to Bloom filter
filter1.add("item1");
filter1.add("item2");
filter1.add(new Integer(17));

// Create a copy of a Bloom filter
BloomFilter filter2 = (BloomFilter) filter1.clone();

Collection c = new HashSet(); // collection should contain non-null items
c.add("item3");

// Add the items contained in collection to the Bloom filter
filter2.addAll(c);

// Compute the intersection of two Bloom filters
BloomFilter intersectionFilter = ((BloomFilter) filter2.clone()).intersect(filter1);

// Check filters for equality
intersectionFilter.equals(filter1); // Should return true
intersectionFilter.equals(filter2); // Should return false

// Compute the union of two Bloom filters
BloomFilter unionFilter = ((BloomFilter) filter2.clone()).unite(filter1);

//Check filters for equality
unionFilter.equals(filter2); // Should return true
unionFilter.equals(filter1); // Should return false

// Compute the union of two Bloom filters and store the result in the first Bloom filter
filter1.unite(filter2);

// Check filters for equality
filter1.equals(filter2); // Should return true

// Convert the Bloom filter to the string representation
String serialized = filter1.getSerialized();

// Restore the Bloom filter from the string representation
filter2 = new BloomFilter(serialized);

// Check filters for equality
filter1.equals(filter2); // Should return true

// Convert the bit set of the Bloom filter to the string representation
String bitSetString = filter1.getSerializedBitSet();

// Convert the hash function family to the string representation
String hashFamilyString = filter1.getHashFunctionFamily().getSerialized();

// Restore hash function family from the string representation
HashFunctionFamily family = new DefaultHashFunctionFamily(hashFamilyString);

// Restore Bloom filter
BloomFilter filter5 = new BloomFilter(bitSetString, family);
```

5.Future Enhancements

- Develop synchronized version intended for use in multi-threaded environment.
- Develop more sophisticated *HashFunctionFamily* implementation