Ultimate Sets and Maps for Java, Part II

This post is a second part of Ultimate Sets and Maps for Java. In the first part, we discussed memory-efficient implementations of sets and maps. These data structures efficiently support *contains(key)* operation. In this part of the article, we discuss more advanced querying:

- How to efficiently test that a collection of items meets a filtering criteria *contains(key1) AND contains(key2) AND …. AND contains (keyN)*
- How to efficiently test that a collection of items meets a filtering criteria (*contains(key11) OR contains(key12) OR …) AND (contains(key21) OR contains(key22) OR …) AND …*
- How to efficiently eliminate duplicates from a collection of items?

Querying Sets

In this section, we discuss two structures: Bloom filter and Reflector. These structures are useful when one has a number of collections (e.g. documents) and need to select all collections that meet a filtering criteria which is a boolean expression over *contains(key)* predicates.

The Bloom filter can be used for filters like *contains(key1) AND contains(key2) AND …. AND contains (keyN)*, i.e. for containsAll() operation.

Reflector is a structure that designed for filters like (*contains(key11) OR contains(key12) OR …) AND (contains(key21) OR contains(key22) OR …) AND …*

Bloom Filter

The Bloom filter is a probabilistic data structure that is used in some caching systems and databases. We briefly describe the main properties of the Bloom filter, more information can be found on Wikipedia and in Mitzenmacher's Probability and Computing .

- The nature of the Bloom filter is close to the hash table. For any element it generates a hash – a *signature* – which is simply an array of bits. Signature of a set of elements is a bitwise OR of signatures of elements.
- Let us have a set A of elements and filter F which is also a set of one or more elements. We want to check if A.containsAll(F). In order to do this, we generate signatures s_A and s_F, for A and F, respectively. If s_A contains 1's at all positions that correspond to signatures of elements from F, i.e. s_A && s_F == s_F, then this testifies that A contains F.

- Testing by means of signatures *does not* guarantee a correct answer, because we operate with hashes which are inherently eligible for collisions. If A doesn't contains all elements from F then the Bloom filter can testify that s_A passed filter s_F, i.e. *false positives* are possible. Nevertheless, *true negatives* are not possible. If we need a correct answer then we need to re-check result provided by the Bloom filter. In other words, the Bloom filter partially sorts out sets that do not meet the filter F and, consequently, decreases the number of direct checks.

  Efficiency of the Bloom filter (probability of false positives) depends on lengths of signature and a number of signed elements. We provide a simple implementation of the Bloom filter with 64-bit signatures that can be used for efficient signing of about 10 elements:

```
1    public class BloomFilter64 {
2        private int capacity;
3        private int k;
4        private int maxLoops;
5
6        public BloomFilter64(int capacity) {
7            this.capacity = capacity;
8            k = (int)(64.0 / capacity * Math.log(2)); // theoretically optimal formula for weight of
9    signature
10           maxLoops = k * 10;
11       }
12
13       public long signature(int[] elements) {
14           long filter = 0;
15           for(Integer e : elements) {
16               filter |= signature(e);
17           }
18           return filter;
19       }
20
21       private long signature(int e) {
22           long fingerprint = 0;
23           int i;
24           for(i = 0; i < maxLoops && Long.bitCount(fingerprint) != k; i++) {
25               fingerprint |= (1L) << (hash(e+i) % 64);
```

```
        }
        if(i == maxLoops) {
            throw new IllegalStateException("Failed to generate signature");
        }
        return fingerprint;
    }

    private int hash(int e) { // Robert Jenkins' 32 bit integer hash function
        e = (e+0x7ed55d16) + (e<<12);
        e = (e^0xc761c23c) ^ (e>>19);
        e = (e+0x165667b1) + (e<<5);
        e = (e+0xd3a2646c) ^ (e<<9);
        e = (e+0xfd7046c5) + (e<<3);
        e = (e^0xb55a4f09) ^ (e>>16);
        return e;
    }

    public boolean test(long filter, long signature) {
        return (filter & signature) == signature;
    }
}
```

Usage of BloomFilter64 is demonstrated in the following snippet:

```
BloomFilter64 bloom = new BloomFilter64(10); // it is expected that filter will sign about 10 elements

long signature = bloom.signature(new int[] {1, 56, 87, 2345, 92}); // sign 5 elements
long filterA  =  bloom.signature(new int[] {56, 87});
long filterB  =  bloom.signature(new int[] {87, 2345, 777});

bloom.test(signature, filterA); // return true - signature contains 56 and 87
bloom.test(signature, filterB); // return false - signature doesn't contain 777
```

Let us evaluate performance of the Bloom filter. We generate a collection of sets, each one contains setSize elements that are randomly selected from the set of random integers of size elementPoolSize. Then we create a filter randomly selecting one of the sets and truncating it to filterSize elements. Obviously, the longer the filter the higher its selectivity. We measure how much time does it take to iterate over all sets and test each one against the filter using the following two approaches:

- Both sets and filter are stored as HashSet<Integer> and testing is performed via HashSet#containsAll() operation.
- Each set and filter is associated with a signature generated via BloomFilter#signature. We first test set's signature against filter's signature using BloomFilter#test()operation and, if passed, we test the set against the filter using HashSet#containsAll() to filter out potential false positives.

  We repeat the experiment multiple times and average measurements over different filters and sets.

  In this test schema, we can vary filter selectivity via elementPoolSize and filterSize parameters for fixed setSize. For example, if setSize=10, filterSize=1, and elementPoolSize=20 then selectivity will be about 50% because filter is selected from 20 possible values and each 10-element set contains one half of possible values. Let us choose relatively small elementPoolSize=20 and setSize=10 to achieve a comprehensive range of selectivities and measure throughput (filtrations/sec) of both techniques for different filterSize:

Filter size (elements)	True filter selectivity (%)	Bloom filter selectivity (%)	HashSet throughput (filtrations/sec)	Bloom Set throughput (filtrations/sec)	BloomSet/HashSet throughput ratio
1	50.01528	50.63274	3417634	5217845	1.53 x
2	23.69485	28.23952	2857142	6402048	2.24 x
3	10.53322	15.63630	2871088	12113870	4.22 x
4	4.33940	7.75789	2714809	19704433	7.26 x
5	1.62736	3.94698	2675585	32626427	12.19 x

10	0.00057	0.02942	2688172	173913043	**64.70 x**

One can see that Bloom filter is a very effective technique comparing to the standard hash table. Nevertheless, it's efficiency depends on many factors (variation of set sizes, cost of false positives etc) and one should carefully adjust parameters to achieve good results.

Reflector

Reflector is a simple structure for testing sets with filters like (*contains(key11) OR contains(key12) OR …) AND (contains(key21) OR contains(key22) OR …) AND …*. The basic idea is that we have a bit flag for each OR'ed group and map each attribute to the corresponding flag. So, if the filter passes then all flags are set because each group in the filter toggles a dedicated flag. We introduce the following constraints to keep the implementation simple:

- Maximum number of OR'ed groups is 63. This allows us to use single long variable for flags storage.
- Elements of set and filter (attributes) are positive integers in range (1, N] and N is so small that it's possible to allocate array of N integers.
- Groups cannot have common elements, i.e. a particular attribute appears in the filter only once.

These constraints can be easily removed if necessary: long variable can be replaced by BitSet and hash table can be used if N is large. Let us take a look at the implementation:

```
1    public class Reflector {
2        private long[] index;
3        private int groupNumber;
4
5        public Reflector(int maxElement, int[][] filter) {
6            index = new long[maxElement];
7            for(int g = 0; g < filter.length; g++) {
8                for(int e = 0; e < filter[g].length; e++) {
9                    index[filter[g][e]] = 1 << (g + 1);
10               }
11           }
12           groupNumber = filter.length;
13       }
```

```
14
15      public boolean test(int[] set) {
16          long footprint = 0;
17          for(int i = 0; i < set.length; i++) {
18              footprint |= index[set[i]];
19          }
20
21          return BitUtils.testLSB(footprint, groupNumber);
22      }
23  }
24
25  class BitUtils {
26      private static long[] lsbMask;
27
28      static {
29          lsbMask = new long[64];
30          for(int i = 1; i < 64; i++) {
31              lsbMask[i] = lsbMask[i-1] | (1 << i);
32          }
33      }
34
35      public static boolean testLSB(long value, int bits) {
36          return (value & lsbMask[bits]) == lsbMask[bits];
37      }
38  }
```

- Refelector instance is created for each filter and can be used to test (filter) multiple sets.
- Filter is passed to the constructor as an array of arrays, each inner array represents an OR'ed group. For example, filter (1 OR 3) AND (6 OR 7 OR 8) can be created as **new int[][] {new int[]{1,3}, new int[]{6,7,8}}**
- BitUtils.testLSB(v, k) method returns true if k less significant bits are all ones in v and false otherwise.

  It's clear that Reflector is very efficient in case of small sets and complex filters because its performance does not depend on size of the filter (number of OR'ed groups and size of each group).

  Duplicates Removal

In this section we discuss how to efficiently process a bunch of collections removing duplicates from each of them. There are different approaches to this problem:

- Pass each collection through hash table.
- If the total number of possible elements is predefined then it is possible to allocate a bit set where each position represents one element and traverse collection setting corresponding bit to 1 for each element.
- Sort each collection and traverse sorted list excluding duplicates.

StackSet is a data structure that combines the first two approaches eliminating overheads on allocation and cleanup of temporary memory. StackSet can be described as follows:

- StackSet is the open addressing hash table where all elements are linked in order of their insertion by means of pointers.
- add() method inserts an element to the table and target a corresponding pointer to the previous element. lastWritePosition points to the last inserted element.
- poll() method returns an element which is pointed by lastWritePosition, cleans its position, and switches lastWritePosition to the previous element.
- if one inserts several elements and polls StackSet till the EMPTY state then StackSet instance is empty and ready to be reused.

```
1   public class StackSet {
2
3       private int elements[];
4       private int pointers[];
5
6       private int lastWritePosition = -1;
7       private int addedElements = 0;
8       private int capacity;
9
10      public static final int EMPTY = -1;
11
12      private static final int FREE = -1;
13
14      public StackSet(int capacity, double loadFactor) {
15          this(MathUtils.nextPrime((int)(capacity / loadFactor)));
16          this.capacity = capacity;
17      }
18
19      private StackSet(int size) {
```

```
20          elements = new int[size];
21          Arrays.fill(elements, FREE);
22          pointers = new int[size];
23          this.capacity = size;
24      }
25
26      public void add(int key) {
27          int position = indexOfKey(key);
28
29          if(position >= 0) {
30              if(addedElements > capacity) {
31                  throw new IllegalStateException("Set is full");
32              }
33              elements[position] = key;
34              pointers[position] = lastWritePosition;
35              lastWritePosition = position;
36              addedElements++;
37          }
38      }
39
40      public int poll() {
41          if(lastWritePosition < 0) {
42              addedElements = 0;
43              return EMPTY;
44          }
45          int result = elements[lastWritePosition];
46          elements[lastWritePosition] = FREE;
47          lastWritePosition = pointers[lastWritePosition];
48          return result;
49      }
50
51      private int indexOfKey(int e) {
52          final int length = elements.length;
53
54          int startPosition = e % length; // the first hash function
55          int decrement = 1 + (e % (length-2)); // the second hash
56  function
57
```

```
58          while (elements[startPosition] != e && elements[startPosition] !=
59  FREE) {
60              startPosition -= decrement;
61              if (startPosition < 0) {
62                  startPosition += length;
63              }
64          }
65
66          if(elements[startPosition] == FREE) {
67              return startPosition;
68          } else {
69              return -1;
70          }
        }
    }
```

Usage of StackSet is illustrated below:

```
1   StackSet stackSet = new StackSet(capacity, loadFactor);
2   for(int[] array : arrays) {
3     for(int i = 0; i < array.length; i++) {
4       stackSet.add(array[i]);
5     }
6     int numberOfDistinctElements = 0;
7     while(stackSet.poll() != StackSet.EMPTY) {
8       numberOfDistinctElements++;
9     }
10    // numberOfDistinctElements counted the number of distinct
11  elements in array
    }
```

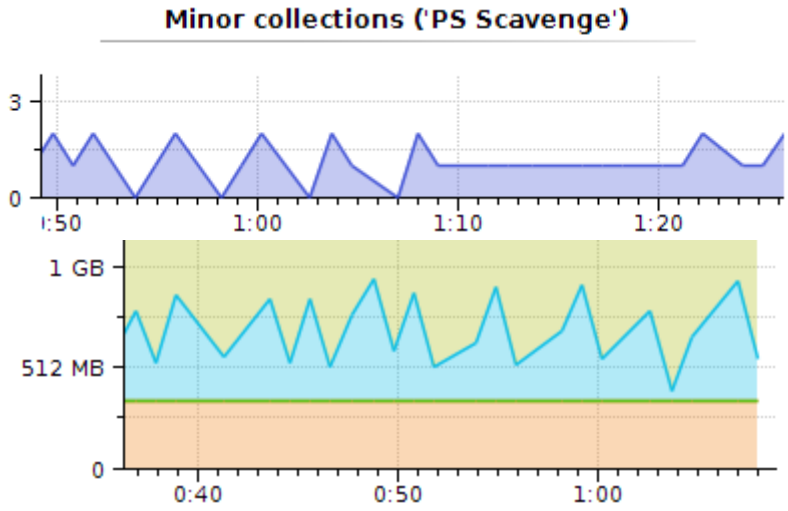
Note that stackSet is created only once and this single instance is used for processing of arrays collection. In this simplified example we need to know the capacity of StackSet (maximum number of distinct elements is array or at least maximum array length) before calculations. In practice StackSet can grow dynamically by means of rehasing as any hash table.

We compare performance of StackSet with HashSet and duplicates removal via sorting. The results for different array sizes and duplicates amount are shown below:

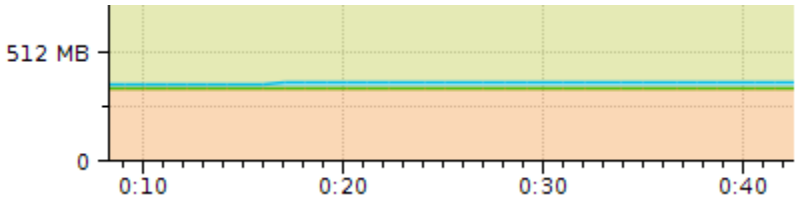
Duplicates	Array Size	HashSet	StackSet	Sorting

(%)	(elements)	throughput (arrays/sec)	throughput (arrays/sec)	throughput (arrays/sec)
90	10	2,085,070	7,194,244	6,361,323
90	100	236,854	798,403	323,572
90	1000	21,121	83,368	22,441
50	10	1,653,986	4,573,519	3,881,987
50	100	191,846	539,956	174,779
50	1000	16,728	59,031	14,507
10	10	1,419,144	4,149,377	3,522,987
10	100	170,619	502,891	173,726
10	1000	15,656	55,509	14,584

We see that StackSet is effective enough, although performance of sorting is very close for small arrays. It is worth noting that HashSet generates a lot of garbage during its work:



whereas StackSet does not allocate dynamic memory at all:



Source: http://highlyscalable.wordpress.com/2012/01/01/ultimate-sets-and-maps-for-java-p2/