Ultimate Sets and Maps for Java, Part I

Some time ago our team had been requested to develop several Java components for structured information retrieval. After the initial research, we concluded that standard approaches like inverted indexes are not well applicable to our problem because of specific business requirements. As a result we faced a necessity to design our own custom indexes and index processors with the following properties:

- Memory footprint of indexes should be as low as possible
- Index processing should be efficient in CPU utilization and memory allocation (produce a small amount of garbage objects)
- Indexes are mainly immutable i.e. performance of index building and modification is not significant
- Indexes should be partitionable i.e. it should be possible to distribute an index across the cluster

The details of our business requirements are irrelevant, as well as a particular structure of the designed indexes, but it's important that we ended up with the following problem statement:

There is a set of documents and each document is a collection of attributes or a group of such collections. It's necessary to store and process all documents one by one (e.g. there exist admixtures of business logic that do not allow to leverage inverted indexes). The following questions are in focus:

- ***How to store sets?*** *It's necessary to store sets of items in immutable collections without duplicates (Set semantics). Small memory footprint is a must, efficient iteration over collection and fast contains() operation are also required. This problem is very close to building compact and immutable Map with fast get() operation (just need to associate each key with a value).*
- ***How to query sets?*** *This generic problem can be described as follows: There is a collection of documents and each document D is a set of attributes {A,B,..,X,Y}. Select all the documents that meet an attribute filter. The filter is an boolean expression that permits (or not) a set of attributes.*
- ***How to build sets?*** *This question is about duplicates removal. Another problem statement is merging of several potentially intersecting sets into the single collection without duplicates.*

A few more assumptions:

- *Collections are designed to store IDs (document IDs, attribute IDs etc), so we focus our attention on primitive types assuming that keys are positive integers.*
- *Most collections are immutable or have immutable capacity, high performance of add() operation is not a goal. The remove() operation is an extremely unusual case, we will simply omit this aspect for the sake of brevity.*

From an implementation perspective, these tasks assumes intensive usage of Maps and Sets, but standard Java collections like HashSet are not always the best choice because of the large memory footprint (objects instead of primitive types) and intensive memory consumption in runtime (creation of new Map.Entry objects etc). The alternative frameworks like Colt look promising, but also don't cover all our needs.
Eventually we developed our own collections. In this article we briefly discuss simplified versions of our structures (there is no need in the detailed analysis because all the algorithms are well known) and present results of performance testing. The main goal is to share the sources of the basic implementations that may be useful for developers who work with custom data structures.

Compact Sets

In this section we discuss several data structures that can be used for compact set storage. Most of them are based on hashing, although this is not the only choice.

Open Addressing Hash Set

In this section we present a typical implementation of open addressing hash table with double hashing. This is a standard algorithm (for example, see Cormen's Introduction in Algorithms) that is widely used in many libraries like fastutil, Trove, or Colt. We provide several comments on implementation without detailed description of the algorithm:

- For simplicity, the capacity is fixed. This doesn't affect the design deeply because capacity can be extended by means of rehashing.
- Addresses are generated using the double hashing technique.
- We use a table keys that has a size always equal to a prime number. This guarantee that hash functions in indexOfKey() iterate all the table searching FREE position for a new element.
- MathUtils.nextPrime(m) returns a prime number that is greater or equal than m. Implementation is shown in the Appendix A.
- Typical load factor is 0.5. This means that we need to allocate array of 2n integers to store n elements. This is pretty compact. Moreover, the structure typically performs

well for the load factors of about 0.7–0.9, although the greater load factor, the worse the performance.

```
1     public class OpenAddressingSet {
2        protected int keys[];
3        protected int capacity;
4        protected int size;
5
6        protected static final int FREE = -1;
7
8        public OpenAddressingSet(int capacity, double loadFactor) {
9           this.capacity = capacity;
10          int tableSize = MathUtils.nextPrime( ((int)(capacity / loadFactor)) );
11
12          keys = new int[tableSize];
13          Arrays.fill(keys, FREE);
14       }
15
16       public boolean contains(int key) {
17          return indexOfKey(key) >= 0;
18       }
19
20       public boolean add(int key) {
21          if(size >= capacity) {
22             throw new IllegalArgumentException("Set is full");
23          }
24
25          boolean result;
26          result = addInternal(key);
27          if(result) {
28             size++;
29          }
30          return result;
31       }
32
33       protected boolean addInternal(int key) {
34          int position = indexOfKey(key);
```

```
35
36          boolean added = false;
37          if(position < 0) {
38              position = -position-1;
39              added = true;
40          }
41
42          keys[position] = key;
43
44          return added;
45      }
46
47      private int indexOfKey(int key) {
48          final int length = keys.length;
49
50          int startPosition = key % length; // the first hash function
51          int step = 1 + (key % (length-2)); // the second hash function
52
53          while (keys[startPosition] != key && keys[startPosition] != FREE) {
54              startPosition -= step;
55              if (startPosition < 0) {
56                  startPosition += length;
57              }
58          }
59
60          if(keys[startPosition] == FREE) {
61              return -startPosition-1;
62          }
63
64          return startPosition;
65      }
66  }
```

Performance of contains() operation may be increased by additional reordering of elements during the insertion. One possible approach is so-called Brent algorithm. It is not widely used, but Knuth provides a sufficient description in The Art of Computer Programming Vol 3. The algorithm is quite complex, so we simply provide the implementation (variables are named in accordance with Knuth's notation):

```
1   public class BrentOpenAddressingSet extends OpenAddressingSet  {
2       public BrentOpenAddressingSet(int capacity, double loadFactor) {
3           super(capacity, loadFactor);
4       }
5
6       protected boolean addInternal(int key) {
7           final int length = keys.length;
8
9           int h1 = key % length; // the first hash function
10          int h2 = 1 + (key % (length-2)); // the second hash function
11
12          int scanStep = 0;
13          int scanCursor = h1;
14          boolean isInserted = false;
15          boolean isPositionFree;
16          while (!isInserted) {
17
18              isPositionFree = keys[scanCursor] == FREE;
19              if(isPositionFree || keys[scanCursor] == key) {
20                  keys[scanCursor] = key;
21                  isInserted = true;
22              } else {
23                  scanCursor -= h2;
24                  if (scanCursor < 0) {
25                      scanCursor += length;
26                  }
27
28                  boolean isOptimized = false;
29                  isPositionFree = keys[scanCursor] == FREE;
30                  if((!isPositionFree && keys[scanCursor] != key) && scanStep >= 2) {
31                      int rank = scanStep - 1;
32                      for(int j = 0; j < rank && !isOptimized; j++) {
33                          int p_j = (h1 - j * h2) % length;
34
35                          if(p_j < 0) {
36                              p_j += length;
37                          }
38
```

```
39                      int c_j = 1 + (keys[p_j] % (length-2));
40                      int p_kj = (p_j - (rank - j)*c_j) % length;
41
42                      if(p_kj < 0) {
43                          p_kj += length;
44                      }
45
46                      isPositionFree = keys[p_kj] == FREE;
47                      if(isPositionFree || keys[p_kj] == key) {
48                          keys[p_kj] = keys[p_j];
49                          keys[p_j] = key;
50                          isOptimized = isInserted = true;
51                      }
52                  }
53              }
54
55              isPositionFree = keys[scanCursor] == FREE;
56              if(!isOptimized && (isPositionFree || keys[scanCursor] == key)) {
57                  keys[scanCursor] = key;
58                  isInserted = true;
59              }
60          }
61
62          scanStep++;
63      }
64
65      return isInserted;
66    }
67  }
```

Cuckoo Hash Set

Open addressing hash tables have very small memory footprint, but performance of the lookup has certain pitfalls. The algorithms described in the previous section jump from one position in the table to another until the requested key is found or a free position is found (i.e. there is no such element in the table – unsuccessful lookup). Under some circumstances, the series of jumps can become too long, especially for unsuccessful lookups. The Cuckoo Hashing offers a trade off between the memory consumption and performance/stability of the lookups. It typically uses two tables

instead of one, but both successful and unsuccessful lookups are performed without iterative jumps (compare contains methods in OpenAddressingSet and CuckooSet). We provide a simple implementation of the Cuckoo Set below. Let us briefly comment the several aspects:

- The proposed implementation uses multiplicative hash functions with "magic" multipliers H1, H2 that are discussed, for example, in The Art of Computer Programming Vol 2 .
- In the open addressing hash tables we don't need to resize the table, given that the capacity is predefined and fixed. In CuckooSet we can't guarantee that insertion will find a place for new element, even after multiple permutations of the existing elements (maximum number of permutations is regulated by insertRounds). So, increase of the table size and rehashing is an integral part of the implementation.

```
1     public class CuckooSet {
2         private int[] keysT1;
3         private int[] keysT2;
4         private int tableLength;
5         private int hashShift;
6         private int rehashCounter = 0;
7
8         private double loadFactor;
9         private int capacity;
10        private int insertRounds;
11
12        private static final long H1 = 2654435761L;
13        private static final long H2 = 0x6b43a9b5L;
14        private static final long INT_MASK = 0x07FFFFFFF;
15
16        private static final int FREE = -1;
17
18        public CuckooSet(int capacity, double loadFactor, int insertRounds) {
19            this.loadFactor = loadFactor;
20            this.insertRounds = insertRounds;
21
22            initializeCapacity(capacity, loadFactor);
23        }
24
25        public boolean add(int key) {
```

```
26          if(contains(key)) {
27              return false;
28          }
29          for(int i = 0; i < insertRounds; i++) {
30              int h1 = h1(key);
31              int register = keysT1[h1];
32              keysT1[h1] = key;
33              if(register == FREE) {
34                  return true;
35              }
36              key = register;
37
38              int h2 = h2(key);
39              register = keysT2[h2];
40              keysT2[h2] = key;
41              if(register == FREE) {
42                  return true;
43              }
44              key = register;
45          }
46
47          rehash();
48
49          return add(key);
50      }
51
52      public boolean contains(int key) {
53          return keysT1[h1(key)] == key || keysT2[h2(key)] == key;
54      }
55
56      public int getRehashCounter() {
57          return rehashCounter;
58      }
59
60      private void rehash() {
61          int[] oldT1 = keysT1;
62          int[] oldT2 = keysT2;
63
```

```
64          int oldTableLength = tableLength;
65          initializeCapacity(capacity * 2, loadFactor);
66
67          for(int i = 0; i < oldTableLength; i++) {
68              if(oldT1[i] != FREE) {
69                  add(oldT1[i]);
70              }
71              if(oldT2[i] != FREE) {
72                  add(oldT2[i]);
73              }
74          }
75
76          rehashCounter++;
77      }
78
79      private void initializeCapacity(int capacity, double loadFactor) {
80          this.capacity = capacity;
81          tableLength = (int)(capacity / loadFactor);
82          hashShift = 32 - (int)Math.ceil(Math.log(tableLength) / Math.log(2));
83
84          keysT1 = new int[tableLength];
85          keysT2 = new int[tableLength];
86
87          Arrays.fill(keysT1, FREE);
88          Arrays.fill(keysT2, FREE);
89      }
90
91      private int h1(int key) {
92          return ((int)( ((int)(key * H1) >> hashShift) & INT_MASK) ) % tableLength;
93      }
94
95      private int h2(int key) {
96          return ((int)( ((int)(key * H2) >> hashShift) & INT_MASK) ) % tableLength;
97      }
98  }
```

Array Set

We have already discussed several hash-based approaches for sets representation. To have a complete picture, we should mention another one very simple technique.

The ArraySet is a straightforward implementation of set – it stores keys in the sorted array, contains() is implemented as a binary search. The shortcomings are obvious: inefficient add() operation and relatively low performance of binary search for large arrays. However, we can expect good performance and excellent memory footprint for small sets.

```
1     public class ArraySet {
2         private int[] array;
3         private int size = 0;
4 
5         public ArraySet(int capacity) {
6             array = new int[capacity];
7             Arrays.fill(array, -1);
8         }
9 
10        public boolean add(int key) {
11            int index = Arrays.binarySearch(array, 0, size, key);
12            if (index < 0) {
13                int insertIndex = -index-1;
14 
15                if(size < array.length - 1) {
16                    if(insertIndex < size) {
17                        System.arraycopy(array, insertIndex, array, insertIndex + 1, size - insertIndex);
18                    }
19                    array[insertIndex] = key;
20                } else {
21                    int[] newArray = new int[array.length + 1];
22                    System.arraycopy(array, 0, newArray, 0, insertIndex);
23    System.arraycopy(array, insertIndex, newArray, insertIndex + 1, array.length - insertIndex);
24                    newArray[insertIndex] = key;
25                    array = newArray;
26                }
27 
28                size++;
29                return true;
30            }
31            return false;
32        }
33 
```

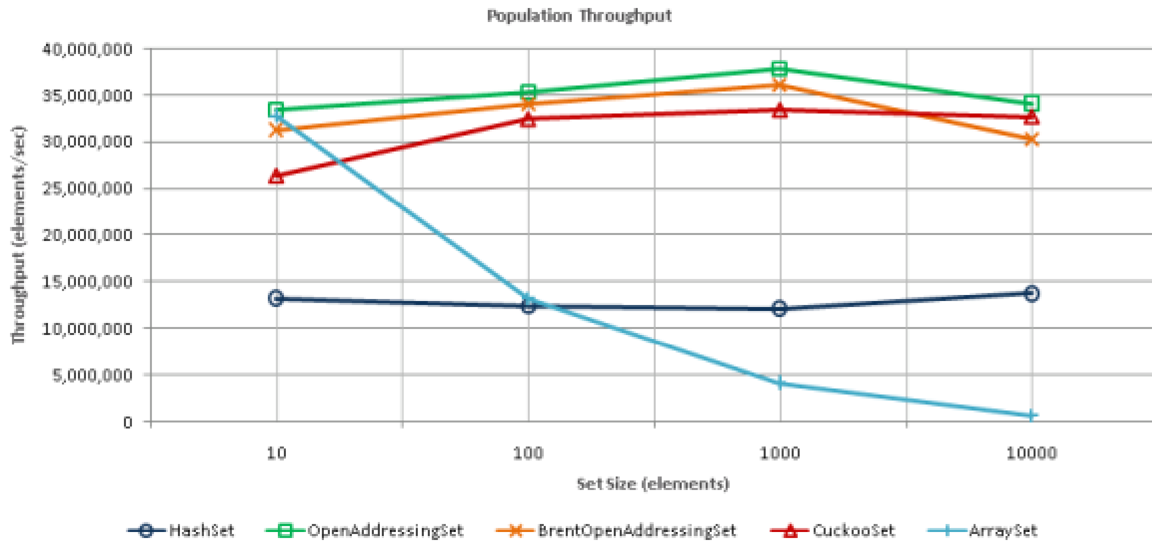
```
34        public int get(int position) {
35            return array[position];
36        }
37
38        public int size() {
39            return size;
40        }
41
42        public boolean contains(int key) {
43            return Arrays.binarySearch(array, key) >= 0;
44        }
45    }
```

Performance Evaluation

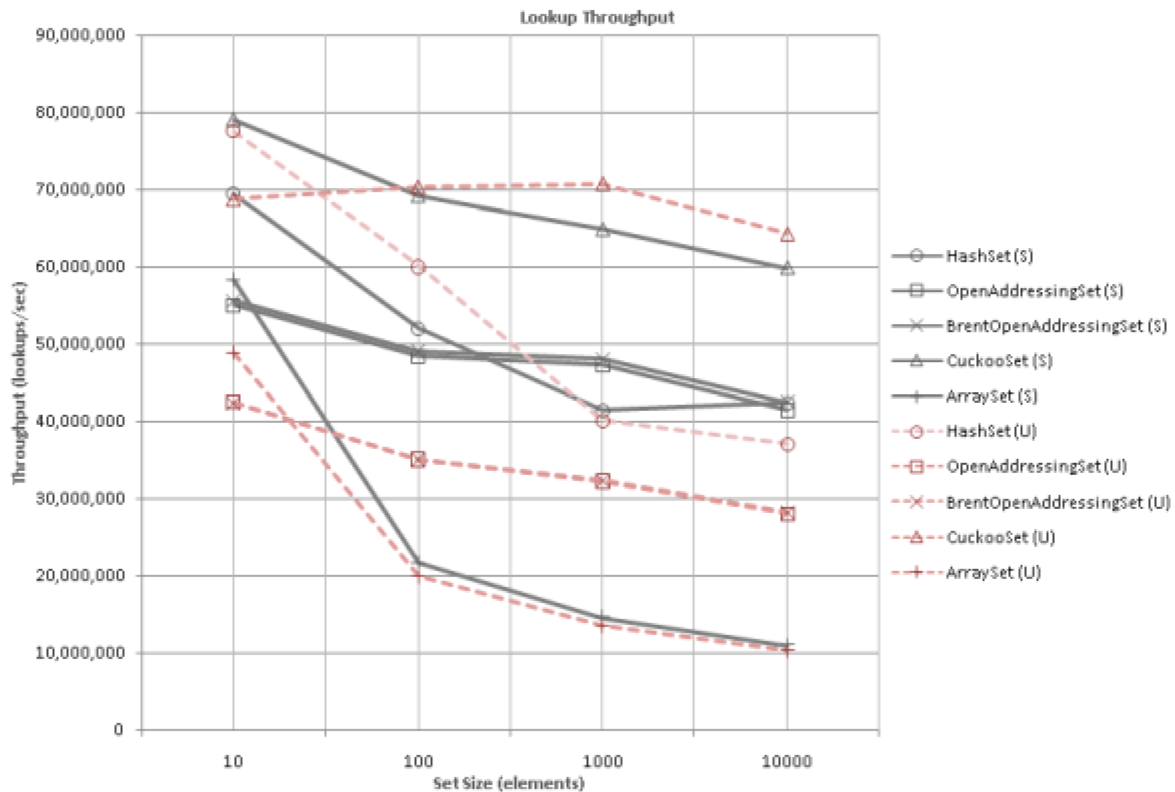
We start with a couple of words about testing environment. We use 64-bit JDK 1.6 u20 under Ubuntu Linux 9.10, 2.93GHz Intel Core Quad processor. All performance tests were executed in a single-thread mode. The absolute values of throughput may be used to understand the order of magnitudes, but it's more correct to consider these values as abstract points that can be compared one with other.

Our primary goal was to design sets with the small memory footprint, so we compare sizes of all the structures. In the HotSpot JVM size of OpenAddressingSet orBrentOpenAddressingSet is about elements * 4 / load factor bytes, size of CuckooSet is about 2 * elements * 4 / load factor bytes, size of ArraySet is about elements * 4. This means that for 10000 elements ArraySet uses 40kB, OpenAddressingSet and CuckooSet use 80kB and 160kB, respectively, for load factor of 0.5. Size of HashSet is about 610kfor 10000 integer keys.

Let us evaluate performance of the described implementations on random keys. First, we measure performance of the set population (add operation) for different set sizes. In this test, memory for all structures is preallocated and the load factor for HashSet is 0.75 (default). For other implementations load factors is 0.5. It's clear that OpenAddressingSet and CuckooSetsignificantly outperform HashSet under these circumstances:



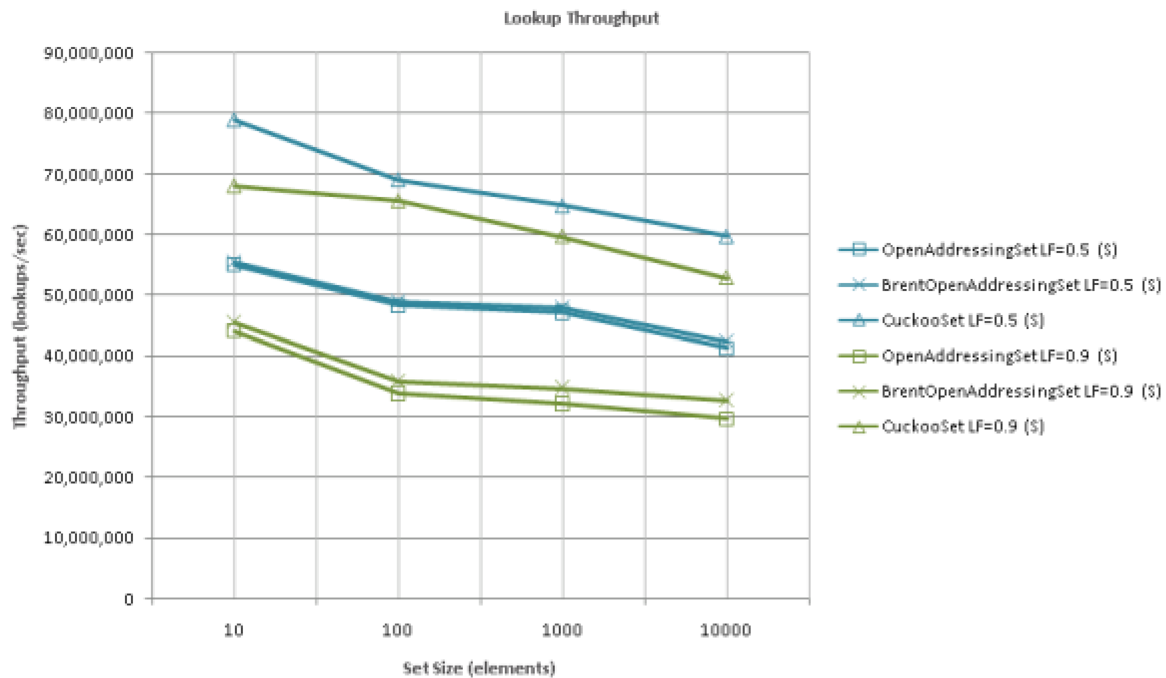
Performance of the lookup operation is depicted below. We test successful (requested key is in the set, marked with (S)), and unsuccessful (key is not in the set, marked with (U)) lookups separately because most algorithms are sensitive to this.



One can see that OpenAddressingSet and CuckooSet work well, but BrentOpenAddressingSet provides almost no gain. It's also clear that ArraySet is very good solution for small sets.

Finally, we compare performance of OpenAddressingSet and CuckooSet for load factors 0.5 and 0.9. Here we see

that BrentOpenAddressingSet outperforms OpenAddressingSet for high load factors:



Appendix A

```
1    public class MathUtils {
2        public static int nextPrime(int n) {
3            int candidate = n;
4            while( !isPrime(candidate) ) {
5                candidate++;
6            }
7            return candidate;
8        }
9
10       public static boolean isPrime(int n) {
11           if((n % 2) == 0 || n % 3 == 0) {
12               return n == 2 || n == 3;
13           }
14
15           int upperSearchLimit = (int)Math.ceil(Math.sqrt(n));
16           for(int i = 6; i-1 <= upperSearchLimit; i += 6) {
17               if(n % (i-1) == 0 || n % (i+1) == 0) {
18                   return false;
19               }
20           }
21           return true;
```

```
22        }
23    }
```

Source: http://highlyscalable.wordpress.com/2011/12/29/ultimate-sets-and-maps-for-java-p1/