Bit Sets

Henry Z. Lo

June 3, 2014

1 Set Operations

Common operations on sets include:

1. Union (or): what is in either one or both of the sets.
2. Intersection (and): what is in both sets.
3. Symmetric difference (xor): what is in one, but not both sets.
4. Subtraction: what is in the first set, but not the second.

See Figure 1 for a visual.

Suppose we have two sets, of size m and n (let $p = m + n$). How can we calculate these using HashSets and TreeSets?

1.1 HashSet

1. Union (or): iterate over both sets - $O(p)$.
2. Intersection: iterate over first set, hash the values, determine whether or not in the second set - $O(m)$.
3. Symmetric difference (xor): opposite of intersection - $O(m)$.
4. Subtraction: same as exclusive or, but use only one set - $O(m)$.

1.2 TreeSets

1. Union (or): iterate over both trees, get a set of size p, then insert p elements into second tree - $O(m \log n)$.
2. Intersection: iterate first tree, remove elements from second - $O(m \log n)$.
3. Symmetric difference: opposite of intersection - $O(m \log n)$.
4. Subtraction: iterate over first tree, for each find elements in second tree - $O(m \log n)$.

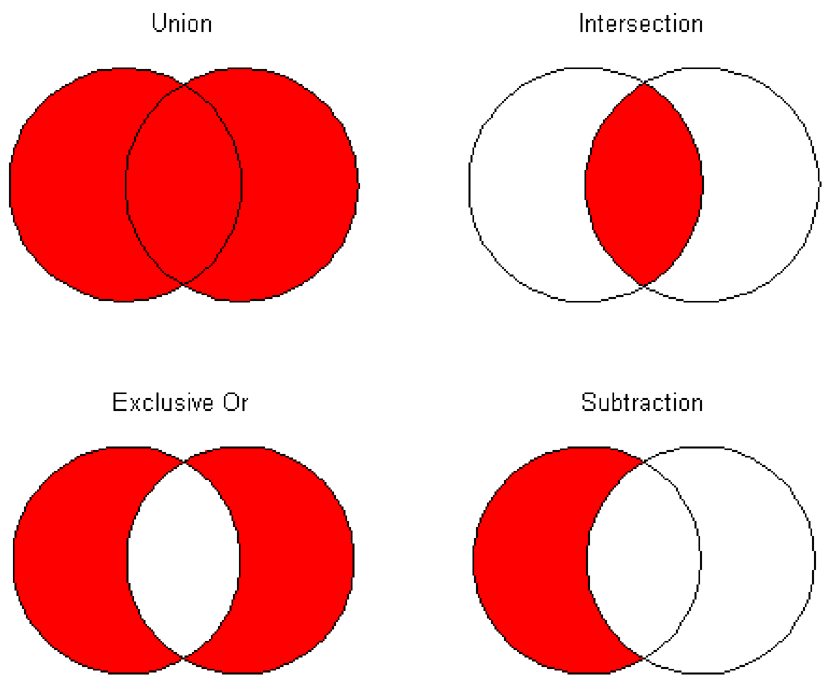

Figure 1: Diagram showing common set operations on two sets

2 Bit Sets

We can use bit strings to represent set membership. Each bit in the string represents either the presence or absence of an element. For example:

$$\{"mouse", "monitor", "computer"\} \rightarrow 1011$$
$$\{"mouse", "computer", "keyboard"\} \rightarrow 1101$$
$$\{"computer", "keyboard"\} \rightarrow 1100$$

For this kind of representation, the length of the bit string must account for every possible element. Also, the location of each bit matters; in this example, the first bit is for "computer", second for "keyboard", third for "monitor", fourth for "mouse".

2.1 Set operations

Using bitwise operations and a bit string representation of two sets, we can easily implement union, intersection, symmetric difference, and subtraction.

Let a and b be two sets represented by bit strings. Then:

- union: a `or` b
- intersection: a `and` b
- symmetric difference: a `xor` b
- subtraction: $a - b$

These are all technically $O(n)$, where n is the number of bytes (a and b) must be the same length. However, the growth rate is very slow, even though it is linear. This is because:

- For a 64-bit CPU, bitwise operations operate on 64 bits in a single CPU operation.
- In tree / hash implementations, you have to compare 64 objects with 64 objects, one by one.
- The simplicity of bit string operations is optimized on the CPU.

Technically, these operations are $O(n/w)$, where w is the word size. In practice, bitwise operations are much faster than other implementations, even if they have lower big-O.

2.2 Memory

Bit strings can store n different pieces of data in n/w words - this is about as compact as you can get without compression.

However, note that every possible item in the set must be represented by a bit. What if sets are very sparse? For example, two sentences of English words. There are hundreds of thousands of words, but often sentences only contain a few. In this case, we have to represent the two sentences using very long bit strings, most of which are 0. This is not optimal.