Software Development (CS2500)

Lecture 19: Bloom Filters

M.R.C. van Dongen

November 15, 2010

Contents

1 Introduction

This lecture studies *bitmaps* and *Bloom Filters*. Bitmaps are space-optimal data structures for representing sets. The candidate set members come from an index set $\{0, \ldots, n-1\}$. Here n is the set's maximum capacity. They can answer simple questions about the set. *Bloom Filters* are also used to represent sets. Here the candidate members come from a key set $\{k_0, \ldots, k_{n-1}\}$. For simplicity, assume that $k_i < k_{i+1}$ for $0 \leq i < n-1$. The answers from Bloom Filters are *probabilistic*: not all answers are correct with 100% probability. For large k_{n-1} such sets can no longer be represented in memory as bitmaps. However, Bloom Filters for such sets are small. They require only a few bits per key.

2 Bitmaps

A *bitmap* is a datastructure which represents subsets of a given index set. Here an index set is a set of the form $\{0, \ldots, n-1\}$. The value n is the set's maximum capacity. A bitmap may be represented using a `boolean` array. Unfortunately, Java does not prescribe how represent a `boolean`. Therefore, the JVM may represent one `boolean` as an `int`. Depending on how the JVM represents a `boolean` bit arrays may be heavy on the memory. This is especially true if the index set is large. This is why in practice bitmaps are often represented as `int` arrays.

We have to implement the following operations:

`Bitmap( int capacity )`: Create a bitmap with a given capacity.

`void add( int index )`: Add a given index to the bitmap.

`void remove( int index )`: Remove a given index from the bitmap.

`int size( )`: What is the cardinality of the bitmap?

`boolean contains( int index )`: Determine if the bitmap contains a given index.

Figure 1 depicts a possible implementation which is based on top of a `boolean` array. To simplify the presentation, we don't override the method `toString( )`.

As already mentioned bitmaps are frequently represented as `int` arrays. When representing a bitmap with an `int` array, we represent a bitmap with capacity of n using $\lceil n/32 \rceil$ `int`s. Here $\lceil x \rceil$ means rounding x up to the smallest possible integer which is greater thanor equal to x. The following demonstrates how this is done using integer division.

```Java
private final int[] bits;
private int size;
public Bitmap( int capacity ) {
    bits = new int[ (capacity + Integer.SIZE - 1) / Integer.SIZE ];
    size = 0;
}
```

Here the class constant `Integer.SIZE` is the size of an `int` in bits. We use bit n to represent index n. Socalled *bitwise* operations are used to implement the instance methods. The following demonstrates how to implement `contains( )`. The operators are explained in Table 1. All operands are `int`s. You don't have to know these operations for the written exam. Also you are not supposed to use them for your assignments. The class method `int numberOfTrailingZeros( int number )` from the `Integer` class returns the number of least significant zeros in the twos complement bit representation of `number`.

```Java
final static int SHIFT = Integer.numberOfTrailingZeros( Integer.SIZE );
final static int MASK  = Integer.SIZE - 1;
public boolean contains( int index ) {
    return (bits[ index >> SHIFT ] & (1 << (index & MASK))) != 0;
}
```

```
public class Bitmap {
    private final boolean[] bits;
    private int size;

    public Bitmap( int capacity ) {
        bits = new boolean[ capacity ];
        size = 0;
    }

    public void add( int index ) {
        size += (bits[ index ]) ? 0 : 1;
        bits[ index ] = true;
    }

    public void remove( int index ) {
        size -= (bits[ index ]) ? 1 : 0;
        bits[ index ] = false;
    }

    public int size( ) {
        return size;
    }

    public boolean contains( int index ) {
        return bits[ index ];
    }
}
```

Figure 1: A possible bitmap implementation.

A bitmap with a capacity of n may be represented with $(n+7)/8$ bytes. We can represent all possible 2^n subset configurations. All operations take constant time. All operations are correct: there are no errors.

Clearly using these bitwise operators is efficient in time and memory. Bitmaps are nice if your index sets are of the form $\{0,\ldots,n-1\}$. This is not always a reasonable assumption. Many applications rely on sets U, with $n = |U|$, but $n \ll \max_{u\in U}(u)$. We could still represent them as bitmaps. However, this would require $\max_{u\in U}(u)$ bits. The resulting data structure will no longer fit in to memory. When this happens a lot of time is wasted on swopping. A datastructure that fits in to memory would be *much* faster:

Operator	Example	Result	Description
`a >> b`	`32 >> 4`	2	Shift `a` right by `b` bits. The `b` high bits become the original sign bit of `a`.
`a >>> b`	`-1 >>> 30`	3	Shift `a` right by `b` bits. The `b` high bits become `0`.
`a << b`	`3 << 3`	24	Shift `a` left by `b` bits. Zeros are shifted into the lower positions.
`a & b`	`5 & 3`	1	Bitwise and of `a` and `b`.
`a | b`	`5 | 3`	7	Bitwise or of `a` and `b`.
`a ^ b`	`5 ^ 3`	6	Bitwise exclusive or of `a` and `b`.
`~a`	`~0`	-1	Bitwise complement of `a`.

Table 1: Bitwise operators.

even if the resulting "set" operations are sometimes inaccurate. In the following section we shall study a neat idea which lets us improve on the standard bitmap implementation.

3 Bloom Filters

A *Bloom Filter* is a probabilistic version of a "set". A Bloom Filter requires much less memory. It cannot answer questions about size. You can add, but you cannot remove from the set.[1] It can be used to answer set membership queries: `contains( int key )`? The accuracy of the answer depends on the answer:

`false`: When a Bloom Filter's answer to the query `contains( key )` is `false` then `key` is *definitely* not in the set.

`true`: When a Bloom Filter's answer to the query `contains( key )` is `true` then the answer may not be 100% accurate. Instead, the answer is accurate with a certain probability.

Let u be a member from the set member universe U. Furthermore, let B be a Bloom Filter. Finally, let's assume we "ask" the Bloom Filter whether $u \in B$. There are four possible cases:

- B returns `false` and $u \notin B$. When the Bloom Filter returns `false` it is true with 100% probability that $u \notin B$.
- B returns `false` and $u \in B$. This situation cannot occur.
- B returns `true` and $u \in B$. When the Bloom Filter returns `true` then the answer is not always correct. However, it is correct with a certain probability.
- B returns `true` and $u \notin B$. This is called a *false positive*. The 'positive' refers to the fact that the answer is `true` and the 'false' to the fact that the answer is incorrect. The *false positive rate* is the probability that a false positive occurs.

[1] However, removing is possible with a so-called *Counting Bloom Filter*. Also removing may be simulated with a second Bloom filter that represents removals.

In general we don't know whether $u \in B$ or $k \notin B$. After all, that's why we're using B. When using the filter, we never have to worry when B returns `false`. We only have to be careful that false positives may occur. So when B returns `true`, the result may be incorrect.

3.1 Implementation

The implementation of Bloom Filters is surprisingly simple. Before looking at the Bloom Filter implementation, let's revisit our bitmaps. We may view a bitmap as a key-value system. The keys and values are equal. It is recalled that with perfect hashing there are no collisions. With perfect hashing we could decide set membership of n members using n bits:

- We hash each set members to its hash code.
- The hash codes should be in the range $\{0, \ldots, n-1\}$.
- Use the hash code as an index in the array.

Unfortunately, collisions are very common. Therefore, an n-bit bitmap representation is out of the question.

Let U be the universe of candidate set members. Let $n = |U|$ and let $I = \{0, \ldots, m-1\}$, with $m \geq n$. With one *perfect* hash function, $h_0 : U \to I$ we can decide set membership with a bitmap. Bloom Filters decide set membership with *several* hash functions, $h_i : U \to I$. The hash functions need not be perfect. An empty filter is represented with m bits: each is 0. To add $u \in U$ to the filter, we set Bit $h_i(u)$ to 1 for each hash function $h_i(\cdot)$. Then u is not contained if Bit $h_i(u)$ is equal to 0 for some hash function $h_i(\cdot)$. Otherwise, u is in the set, but false positives may occur.

For example, let's assume $U = \{0, 1, 7\}$. Furthermore, let's use 4 bits and 2 hash functions.[2] The hash functions are given by $h_0(k) = k + 1 \bmod 3$ and $h_1(k) = k \bmod 4$. Table 2 lists the images of the keys under the hash functions. Column 'Union' in the table lists the bitwise or of setting Bit $h_0(k)$ and Bit $h_1(k)$. Table 3 depicts all possible combinations of subsets of U and the bits which are set if 4 bits are

k	$h_0(k)$	$h_1(k)$	**Union of Bits**
0	1	0	`0011`
1	2	1	`0110`
7	2	3	`1100`

Table 2: Images of keys under hash functions.

used to represent the Bloom Filter.

3.2 Applications

This section briefly discusses some applications of Bloom Filters.

[2] Notice that with a bitmap and no hash function we would need at least 8 bits.

Keys Added	Bits Set	$0 \in B$?	$1 \in B$?	$7 \in B$?
$\{\}$	0000	−	−	−
$\{0\}$	0011	+	−	−
$\{1\}$	0110	−	+	−
$\{7\}$	1100	−	−	+
$\{0,1\}$	0111	+	+	−
$\{0,7\}$	1111	+	+	+
$\{1,7\}$	1110	−	+	+
$\{0,1,7\}$	1111	+	+	+

Table 3: Representing the set $U = \{0,1,7\}$ using a 4-bit Bloom Filter and hash functions $h_0(k) = k \bmod 4$ and $h_1(k) = k + 1 \bmod 3$. There is one false positive for the question $1 \in B$ with 0 and 7 in B.

3.2.1 Key-Value Storage Systems

One application of Bloom Filters are key-value storage systems. These systems use several, *slow* secondary media to store values of keys. Not all candidate keys correspond to values.

For simplicity, let's assume there's only one slow disk. We want to know if a value (and if so which) has the key k. The query B`.contains(` k `)` may result in three cases:

1. B returns `false`. When this happens disk access is avoided: we can trust this answer.

2. B returns `true` and $k \in B$. Disk access is unavoidable. We access the disk to see if there's a value with key k. The value happens to exist and we return the value.

3. B returns `true` and $k \notin B$. Again disk access is unavoidable. We access the disk to see if there's a value with key k. The value doesn't exist and we return $\perp$. Here we return $\perp$ to indicate that there is no value having key k.

If there are enough queries for non-existing keys then this saves time.

3.2.2 Detecting Malicious Websites

A related application is determining whether a given website is malicious. There are billions of websites, so it is impossible to store this information in memory. This application may be viewed as a key-value storage application because we want to check which `boolean` (the value) corresponds to which URL (the key). Let's assume we want to know whether a given URL is malicious. We hash the URL into hash codes h_0, ..., h_{n-1}. If Bit $h_i = 0$ for some i such that $0 \leq i < n$, then the URL isn't know. Since $k \notin B$ the answer is 100% reliable. Therefore *no* URL with hash code k is known as a malicious website. This includes our URL. Otherwise the URL is "known" (but this may be a false positive). There are two possibilities:

- We trust the answer.

- We use a slow database operation to determine if the URL is known.

3.2.3 Probabilistic Spell Checkers

The next example may also be viewed as an example of a key-value storage system: probabilistic spell checkers. Using a Bloom Filter this is easy as π. We start with an empty Bloom Filter, B. For each allowed word, we add the word's hash code to B. If a user enters a word, we compute the hash codes of the word. If some of the hash codes aren't in the filter, then the word is invalid. Otherwise, we assume the word is valid.

3.2.4 Database Joins

The final application which we shall study is pre-processing database joins. Let's assume we have two database tables T_1 and T_2. Let's furthermore assume we wish to compute the join $T_1 \bowtie T_2$. This is an expensive operation, which takes $\mathcal{O}(|T_1| \times |T_2|)$ worst-case time. An important operation is removing redundant rows from T_1 and T_2. These are the rows which cannot contribute to the final result. This also takes $\mathcal{O}(|T_1| \times |T_2|)$ worst-case time. Using Bloom Filters we may remove them in time $\mathcal{O}(|T_1| + |T_2|)$. Of course, the result will contain false positives but we can remove them in a second phase with an accurate algorithm.

Let's remove the redundant rows of T_1. It is recalled that the *scope* of a table are its attributes. Let S be the intersection of the scopes of T_1 and T_1. We start with an empty filter B. There are two phases:

1. In the first phase we add the hash values of the rows in T_2 to B.

2. In the second phase we remove the rows from T_1 whose hash values are not in B.

This works if the hash values of the rows are completely determined by their projections onto S. In practice, this pre-processing step speeds up the overall join computation.

3.3 Properties

In this section we shall discuss some of the properties of Bloom filters.

The time to decide membership is independent of the current "size". With k constant-time hash functions we need $\mathcal{O}(k)$ time. The reliability may be improved by increasing the number of bits. Let R be the *false positive rate*. Here the false positive rate is the probability that a false positive occurs. For ± 4.8 extra bits per member, R reduces by a factor of ± 10. See for example `http://en.wikipedia.org/wiki/Bloom_filter` for a reasonably accessible presentation.

4 Summary

We've studied n-bit bitmaps and m-bit Bloom Filters, where $m \geq n$. Bitmap can decide which numbers in $\{0, \ldots, n-1\}$ are in the set. The i-th member is in the set if and only if the ith bit is set. They are 100% accurate. By adding perfect hashing they can also be used for other kinds of sets.

Bloom Filters are probabilistic data structures. They use several hash functions, $h_i(\cdot)$. A candidate member, u, is not in the filter if some Bit $h_i(u)$ is 0 for some i. Otherwise, Bit $h_i(u)$ is 1 for each i. In this case u is in the set, but false positives may occur.

5 For Wednesday

Study the notes.