Bloom Filter Cascade: Implementation and Performance Analysis

Sungmin Cho

July 14, 2014

1 Executive Summary

A Bloom filter cascade (BFC) is implemented and the performance is evaluated and analyzed. The implementation algorithm is shown and the size efficiency and false positive rates under various conditions are shown and analyzed. We proved that the size for BFC is always larger than that of optimzed Bloom Filter (OBF).

2 Implementations

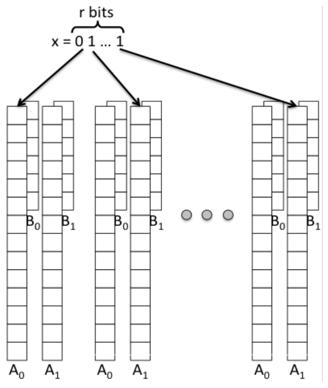
Figure 7(a) shows the structure of BFC which contains parameters to describe its properties. The structure has multiple buffers for r bits of data, each bit has two (1 and 0) buffers. Though only two levels (A and B) of buffers are shown in the drawing, the levels can be n levels in depth. We use m_i to describe the buffer in level i, we may use m_i^1 or m_i^0 to specify if the buffer belongs to 1 or 0 buffer at level i. Alphabets may be used to indicate the buffers, A_1 equals m_0^1, B_0 equals m_1^0, and so on. For each level of buffers, k_i numbers of hash functions are used.

2.1 Creation

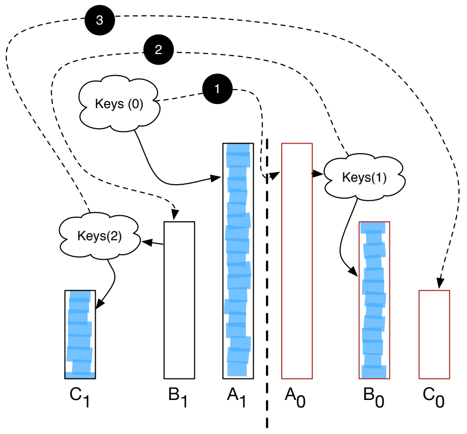
Figure 7(b) shows how the m_i buffers are configured. The diagram shows only one bit out of r bits, and the case when the maximum depth of buffers is 2 (C or m_2).

The first step is to grouping of keys to encode for each bit poisition. Let us say that we encode the dictionary {"00" = 0, "01"=1, "10"=2, "11"=1}. For bit position 0, "00" and "10" becomes a group for 0, then "10" and "11" makes another group for 1. Likewise, for bit position 1, "00" and "01", "10" and "11" makes group for 1 0 and 1 respectively.

After the grouping, keys in group 1 ($Keys(0)$ in figure 7(b)) are used to setup buffer in level 0: A_1 (keys in group 0 are configured the same way, but we explain the case for 1 for simplicity in explanations). $Keys(0)$ should be



(a)



(b)

Figure 1: Bloom Filter Cascade. (a) Architecture (b) Creation

checked if they may cause false positives in A_0 (❶), in this case, the keys that cause false positives ($Keys(1)$) are used to setup B_0.

However, $Keys(1)$ should not cause false positives in B_1. For example, if "10" returns $A_1 = 1, A_0 = 1$, it should return $B_0 = 1$ to indicate that $A_0 = 1$ is false positive with a guarantee that $B_1 = 0$.

Keys in $Keys(1)$ are checked what keys generate false positives (❷), the false positives generators are collected as $Key(2)$ to setup C_1. Likewise, C_0 should be used to check if there is no false positives again (❸). This process goes on until there is no false positives generating keys.

Algorithm 1 describes the mechanism. For simplicity in algorithm, we introduced a *direction* variable that indicates what buffer (0 or 1) is used to check false positives.

In the algorithm, *keys* are data structure to store the keys that described as $Keys(i)$ are stored. $keys(r)$ indicates the keys group at bit position r. Likewise, $tables(r)$ indicates the buffers at bit position r.

These data structures has a method $add(bit, level, key)$ to store key at level $level$ for bit bit, and $set(bit, level)$ to return the keys.

table data structure has $match(bit, level)$ to filter out the keys that generate false positives, and $set(bit, level, key)$ for setting up the buffers.

For each bit of a value, the key set is partitioned into two groups: for the keys that has a bit set and for those unset; algorithm 2 shows the idea how the keys are partitioned for a position i.

2.2 Retrieval

Algorithm 3 explains how the value is retrieved when a key is given. For example, with maxim depth of 2 in buffers, when key "01" returns (1,1) from A and

Algorithm 1: CREATE

```
Input: Dictionary $d$, bit width $r$, arrays $tables$ and $keys$
1  $results = \{\}$
2  for $i$ in $0 \ldots r-1$ do
3      $key1, key0 \leftarrow partition(d, i)$
4      $level \leftarrow 0$
5      for $bit$ in $\{1, 0\}$ do
6          $keys(i).add(bit, level, key\{bit\})$
7          $tables(i).add(bit, level, key\{bit\})$
8      while $keys(i).get(bit = 0, level).size > 0 \wedge keys(i).get(bit = 1, level).size > 0$ do
9          $level \leftarrow level + 1$
10         for $bit$ in $\{1, 0\}$ do
11             $direction \leftarrow (bit + level)\%2$
12             for $key$ in $keys(i).get(bit, level - 1)$ do
13                 if $tables(i).match(direction, level - 1, key)$ then
14                     $tables(i).set(bit = direction, level, key)$
15                     $keys(i).add(bit, level, key)$
```

Algorithm 2: PARTITION

```
Input: Dictionary $d$, index $i$
Output: A tuple of two sets of keys $\{key1, key0\}$
1  $key1 \leftarrow \{\}$
2  $key0 \leftarrow \{\}$
3  for $\{key, value\}$ in $Dictionary$ do
4      if $value[i] = 1$ then
5          $key1.add(key)$
6      else
7          $key0.add(key)$
8  return $\{key1, key0\}$
```

(1,0) from B for bit 0, we can interpret it as (0,1) because B indicates that the first element in (1,1) is false positive.

With maximum depth of 3, the returned values of {(1,1), (1,1), (1,0)} indicates the value (1,0) with the same logic. We can generalize it with exclusive or($\oplus$) operation for all the bits in the first and second element in the returned values from buffers.

When the result is (0,0) (line 4), or we reached the last buffer without resolution (line 12), we declare it as non-data ($\perp$).

Algorithm 3: GET

```
Input: key $key$, bitwidth $r$, index $i$
Output: $value$ or $\perp$
1  $a1 \leftarrow table(i).match(bit = 1, level, key)$
2  $a0 \leftarrow table(i).match(bit = 0, level, key)$
3  if $a1 = false \wedge a0 = false$ then
4      return $\perp$
5  if $a1 = true \wedge a0 = false$ then
6      return $(1, 0)$
7  if $a1 = false \wedge a0 = true$ then
8      return $(0, 1)$
9  if $a1 = true \wedge a0 = true$ then
10     $maxLevel \leftarrow tables(r).getMaxLevel()$
11     if $maxLevel = level$ then
12         return $\perp$
13     $res \leftarrow Get(key, r, i + 1)$
14     if $res = \perp$ then
15         return $\perp$
16     return $(res_1 \oplus 1, res_2 \oplus 1)$
```

3 Results

With the implementation in section 2, we tested the performance of BFC in terms of size efficiency and false positive rates. Table 1 shows the parameters we used for the tests.

For 'buffer patterns', the last size specified used for the rest of buffers. For example, $n/2$ pattern, all of the m_i buffers have the same buffer size $n/2$, and with $n/2, n/4$ pattern, only the A has the size $n/2$ while the rest buffers have the $n/4$ size. All the experiments were implemented with $k_i = 2$. We used $r = 8$ for total size, and $r = 16$ for false positives.

Table 2 and figure 2 shows the results with $|A| = n/2$. In the figure, x axis has buffer level i, and y axis has the size of the buffer (blue and red) or the size of the keys (green). The graph shows only one bit sample from r bits in the 1 case.

Name	Annotation
buffer pattern	The configuration of m_i sizes.
buffer depth	The maximum level of buffers at r bit position
total size	$\sum_r \sum_i (\lvert m_i^0\rvert + \lvert m_i^1\rvert)$
% size	The size overhead compared with OBF
FP rate	False positive rates with total size bits

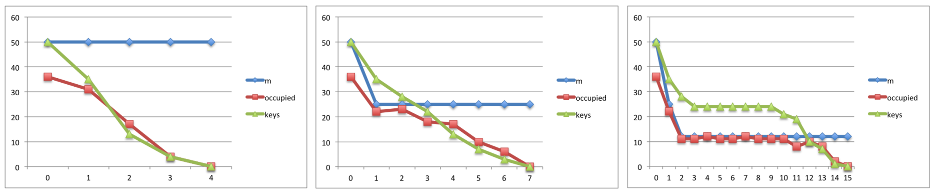
Table 1: Parameters used in the test

Case	buffer pattern	buffer depth	total size	% size[1]	FP rate
1	$n/2$	5	4100	512%	30.03%
2	$n/2, n/4$	8	3650	456%	41.96%
3	$n/2, n/4, n/8$	16	6934	866%	57.18%

Table 2: Table size for $n/2$

Blue line shows the pattern (the size of m_i), and red line shows the number of 1 bits in the buffer. Green line shows the number of keys ($Keys(0), Keys(1) \ldots$) in the previous sections.

When we use less bits for buffers of $i > 0$, the total size increases. The less m_i we use, the more total size we get as a result.



(a) (b) (c)

Figure 2: Case for $n/2$ (a) Pattern 1 (b) Pattern 2

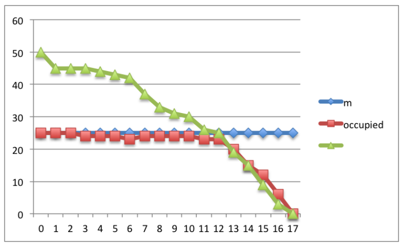
This pattern becomes more distinct with smaller size of m_i. Table 3 and figure 3 shows the case when $|A| = n/4$. What is noticable is the time increase to create the BFC, and the increase in buffer depth. We could not build the BFC with the pattern $n/4, n/8, n/12$, which means the number of false positives generating keys are not decreasing with this configuration.

We could get the smaller total size with $m_0 = n$. By controlling the $|m_{i>1}|$, we could get the maximum total size of 3034 bits from BFC.[2] Table **??** shows various buffer pattern with $m_0 = n$, and its buffer depth, total size, and FP rates.

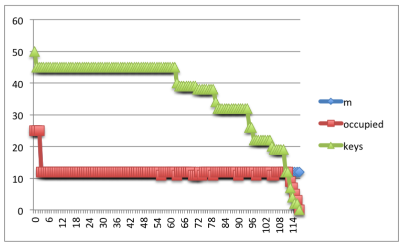
[2]This is not a thorough experiments, so we may be able to get smaller than this size.

Case	buffer pattern	buffer depth	total size	% size	FP rate
1	$n/4$	18	5050	631%	50.66%
2	$n/4, n/4, n/4, n/8$	118	18980	2372%	75.89%
3	$n/4, n/8, n/12$	∞	∞	∞%	N/A

Table 3: Table size for $n/4$



(a)



(b)

Figure 3: Case for $n/4$ (a) Pattern 1 (b) Pattern 2 (c) Pattern 3

3.1 Effects of k

Table 5 and figure 5 shows the various results with different k pattern with $m_0 = n/2, m_{i>0} = n/4$. As the results shows, the total size becomes larger when the k for $m_{i>0}$ becomes smaller.

3.2 Effects of *seed*

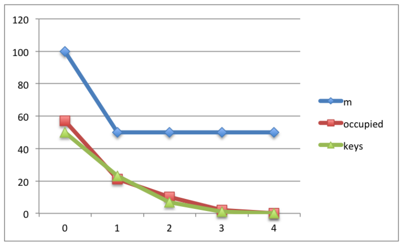
Figure 6 (a) shows the total size with different seed values for hash function. The results shows the total size depends on the choice of *seed* as the variance is 215%. Figure 6 (b) shows the buffer pattern for the minimum size ($seed = 1$), and Figure 6 (c) shows the buffer pattern for the maximum size ($seed = 12$).

4 Analysis

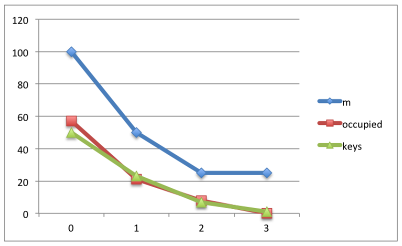
Even though we may be able to have small in $|A|$, the total size is dependent on the sum of all the buffers. As a result, it is practically impossible to get the BFC that is smaller than OBF. This is a proof.

Case	buffer pattern	buffer depth	total size	% size	FP rate
1	$n, n/2$	5	4300	537%	17.29%
2	$n, n/2, n/4$	4	3250	406%	18.04%
3	$n, n/2, n/4, n/8$	4	3034	379%	18.31%
4	$n, n/10$	14	4140	517%	27.36%

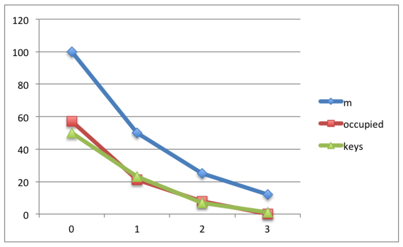
Table 4: Table size for n



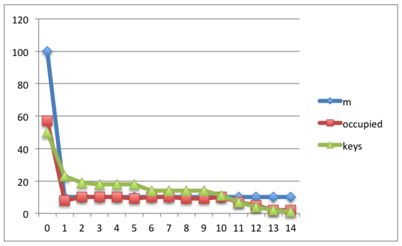
(a)



(b)



(c)

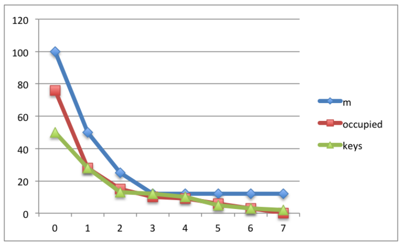


(d)

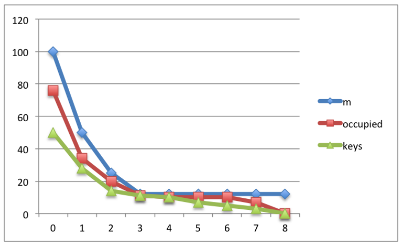
Figure 4: Case for n (a) Pattern 1 (b) Pattern 2 (c) Pattern 3 (d) Pattern 4

Case	k pattern	buffer depth	total size	% size	FP rate
1	$\{3, 2, 2\}$	10	4250	531%	47.48%
2	$\{3, 3, 3\}$	18	6350	793%	48.76%

Table 5: Table size for different k



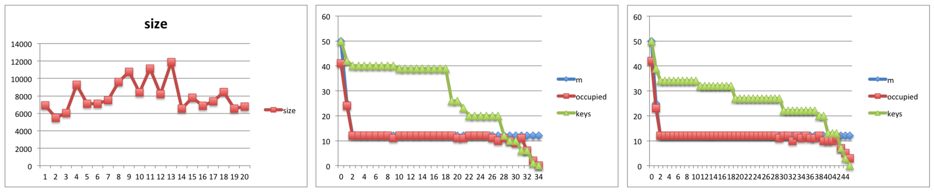
(a)



(b)

Figure 5: Results from different k (a) Pattern for $k = \{3, 2, 2\}$ (c) Pattern for $k = \{3, 3, 3\}$

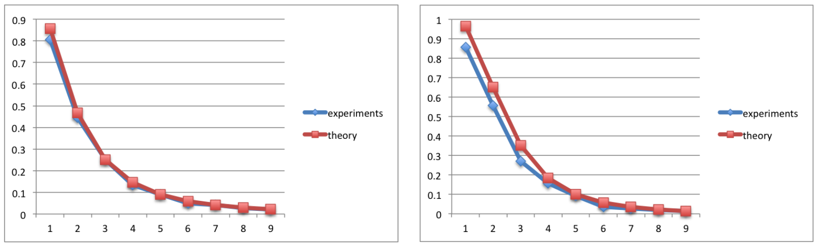
Figure 7 shows the false positive rates of Bloom Filter ($fp = (1 - (1 - 1/m)^{nk})^k$). As the graph shows, with small m the false positive rates are very



(a) (b) (c)

Figure 6: Results from different *seed* (a) size vs. *seed* (b) Pattern for $seed = 1$ (c) Pattern for $seed = 12$

high to cause false negatives in BFC.



(a) (b)

Figure 7: BF false positive rates vs α. (a)k=3. (b)k=5.

Proposition 4.1. *The minimal bit length of a OBF for serialization is* $n \times r + m$.

Proof. For OBF with n inputs, there are n rows in OBF table that are not empty. With r bits for representing the data, the total size of table is $n \times r$. Additionally, m bits of information is needed to indicate which row is empty or not. □

Proposition 4.2. *The bit length of BFC for serialization is always larger than that of OBF.*

Proof. From lemma 4.1, the size of OBF is $n \times r + m$. For BFC size to be smaller than this, the BFC should satisfy two requirements; 1) The size of A_1 and A_0 should be smaller than $n/2$. 2) There should be only level 0 (A_i) buffers.

When requirement 1 is met, the probability of false positives for A_i is $fp = (1 - (1 - 2/n)^{nk})^k$. Furthermore, there should be no false positives for both A_0 and A_1 over all all the r bits. As a result, the probability of no false positives to meet requirement 2 becomes $(1 - fp)^{2r}$ which is practically zero (with $k = 2, n = 100, r = 8$, the probability becomes 4.77×10^{-24}). □

5 Other (possible) issues with BFC

The total size in the previous section is the total bits for the buffers. When serialization/deserialization of BFC, we need configuration data.

Compared to OFC which is rectangular in shape, the shape of BFC is ragged to make it harder to encode as we need to store the different depth for each level. Even worse, each bit has different maximum depths to may cause even larger configuration data. This irregularity might give worse size performance when compressed.

Build time and complexity might be other issue to consider. Based on the the creation and get algorithm, we need recursive methods to build buffers and retrieve data from buffers. In some cases, this may cause some serious delay in processing network data.

Also the high false positives may not be overlooked, from the experiments, the positive rates range between 17.29% and 75.89% with $r = 16$. This implies that we need more bits $(q > r)$ in the input data to reduce the false positives if necessary.