Bitspark

A Scalable Bloom Filter Implementation in Java

January 2020

Raven Computing GmbH

Content

1. What’s a Bloom filter?
2. How does it work?
3. Scalable Bloom filters
4. Our implementation
5. Evaluation
6. Conclusions

Copyright © 2020 Raven Computing

What's a Bloom filter?

A set-like data structure invented by Burton H. Bloom in 1970 [1]

A set is an unordered collection of distinct elements.

In **mathematics** we would write:

$$S = \{ e_1 , e_2 , \dots , e_n \}$$

As the most basic operation one might…

… add element **e** into a set **S**

$$S \cup \{ e \}$$

… query whether a given Element **e** is part of a set **S**

$$e \in S \quad ?$$

Copyright © 2020 Raven Computing

What's a Bloom filter?

A set-like data structure invented by Burton H. Bloom in 1970 [1]

A set is an unordered collection of distinct elements.

In **mathematics** we would write:

$$S = \{ e_1, e_2, \dots, e_n \}$$

As the most basic operation one might…

… add element **e** into a set **S**

$$S \cup \{ e \}$$

… query whether a given Element **e** is part of a set **S**

$$e \in S \quad ?$$

In **Java** we would write (for a hashset of strings):

```java
Set<String> s = new HashSet<>();
```

As the most basic operation one might…

… add element **e** into a set **S**

```java
s.add(e);
```

… query whether a given Element **e** is part of a set **S**

```java
s.contains(e);    ?
```


Copyright © 2020 Raven Computing

What's a Bloom filter?

In programming there is a problem when a set grows too large:

- Every distinct object added to the set must be kept in memory
- That's not practically feasible for a set with millions of elements

The concept of Bloom filters:

Trade memory for accuracy

Copyright © 2020 Raven Computing

How does it work?

A Bloom filter consists of a bit vector of size **n**

Initially all bits are set to zero

In order to insert an element **e** into the filter:

- Use different hash functions to generate **k** hashes of **e**
- Every hash $\mathbf{h_0, \ldots, h_{k-1}}$ represents an index of the bit vector (h_i mod n)
- All bits at the computed indices are set to **1**

In order to query whether an element **e** has been added to the filter:

- Compute **k** hashes of **e** and their respective index (just as when inserting)
- Check whether all bits at the given indices are set to **1**

Copyright © 2020 Raven Computing

Example

Let's say we have a Bloom filter with a bit vector of **15** bits. We use **3** hash functions in this example

$n = 15$
$k = 3$

Copyright © 2020 Raven Computing

Example

Let's add an element $\mathbf{e_0}$ to the filter:

$n = 15$
$k = 3$

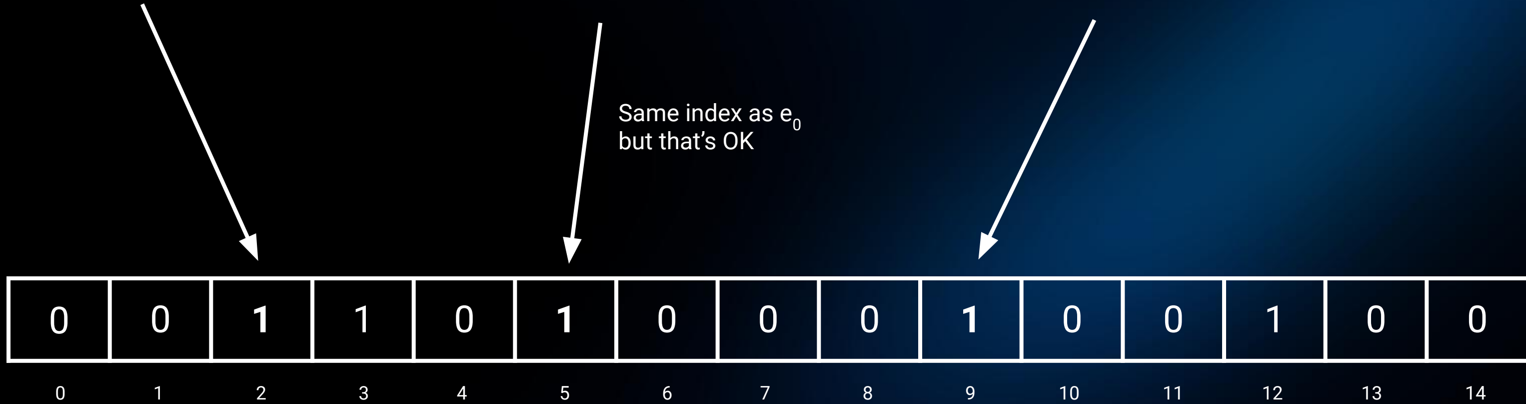
$H_0(e_0) \bmod n = 3$ **$H_1(e_0) \bmod n = 5$** **$H_2(e_0) \bmod n = 12$**

Copyright © 2020 Raven Computing

Example

Let's add another element $\mathbf{e_1}$ to the filter:

n = 15
k = 3

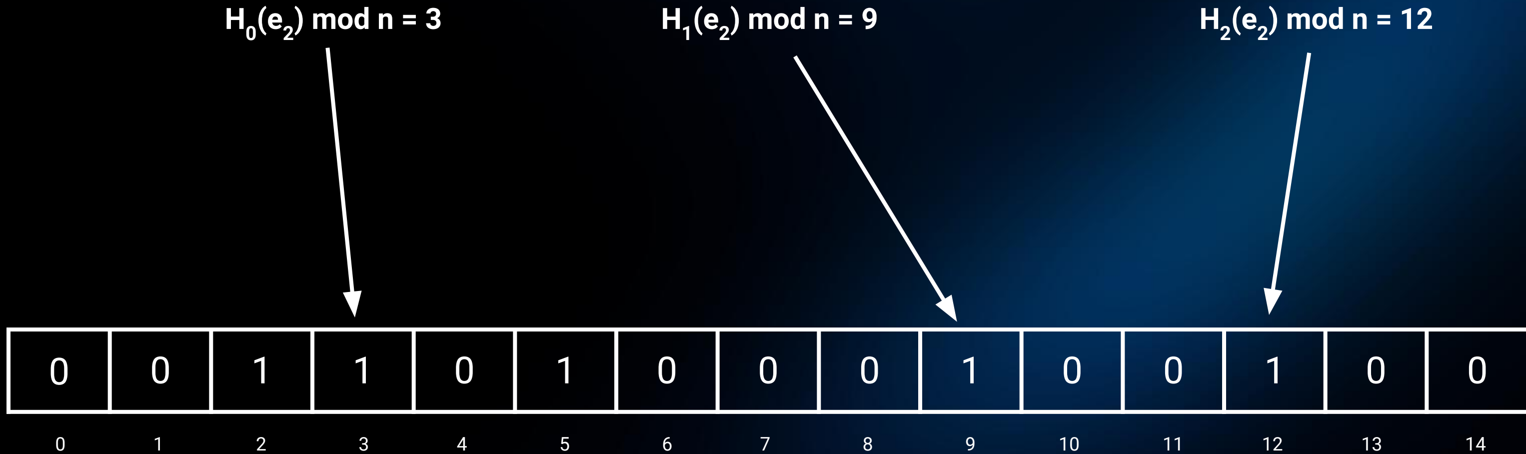


Copyright © 2020 Raven Computing

Example

Let's query the filter to see if element $\mathbf{e_2}$ has been added to the filter: $(e_2 \neq e_0 \neq e_1)$

n = 15
k = 3



Copyright © 2020 Raven Computing

Example

The filter affirms the existence of element $\mathbf{e_2}$ even though it was never added:

- This is called a **false positive**
- It happens when all hashes map to indices of bits which were previously set by insertions of **other** elements

Copyright © 2020 Raven Computing

How does it work?

The **maximum allowed false positive rate** can be specified when the filter is constructed

Problems

- The capacity is fixed and must be specified at the time the filter is constructed
- Once a filter is created it cannot grow dynamically
- Inserting more elements than the initial capacity allows will result in an increasing false positive rate

Copyright © 2020 Raven Computing

Scalable Bloom filters

Based on a paper published by Almeida et al. in 2007 [2]

Scales dynamically during runtime as more elements are added

Is useful when the number of elements cannot be determined at the time the filter is constructed

Adheres to the specified maximum allowed error rate, even when scaling

Advantages

Less overall memory usage

The Bloom filter must not be initialized with the maximum expected number of elements

Copyright © 2020 Raven Computing

Scalable Bloom filters

Divide a filter into **K** separate slices

Elements always set **K** bits when inserting them ⟶ No hash collisions between slices

M = 15

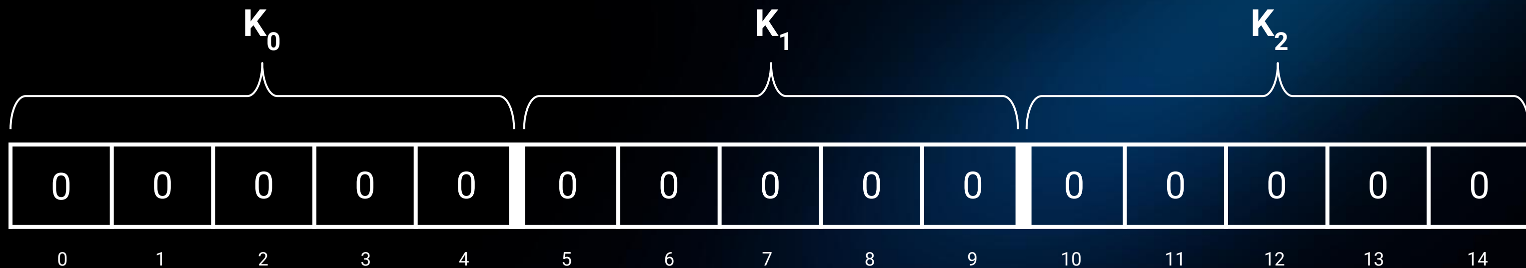
K = 3

m = M / K = 5

When inserting an element **e**, the bits to set are computed for all integers $0 \leq i \leq K$

as:

$$b_i = (H_i(e) \bmod m) + (i\, m)$$



Copyright © 2020 Raven Computing

Scalable Bloom filters

As more elements are added to the filter, more bits get set

The **fill ratio p** is the amount of set bits in relation to the overall vector size

When **p** reaches **50%**, a new vector is added with an increased capacity

M = 15

K = 3

m = M / K = 5

Fill ratio **p** = 8/15 ≈ 53%

The next insertion will cause the filter to grow

Copyright © 2020 Raven Computing

Scalable Bloom filters

When a vector stops being used it becomes read-only and a new one is added

Membership queries are performed by checking all vectors

p = 8/15 ≈ 53%

| 1 0 1 0 1 | 1 0 0 1 1 | 0 1 1 0 0 | K = 3

p = 21/40 ≈ 52%

| 1 1 1 0 0 0 1 1 0 0 | 0 0 0 1 1 1 0 1 1 0 | 0 1 1 0 0 1 0 1 0 1 | 1 0 1 0 0 1 0 1 1 1 | K = 4

p = 26/80 ≈ 32%

| 0 0 0 0 1 1 0 0 0 1 0 1 1 0 0 0 | 0 0 1 0 1 0 1 0 1 0 0 0 1 0 1 0 | 0 0 1 0 0 1 0 0 1 0 1 0 1 0 0 0 | 1 1 0 0 0 1 0 0 0 0 0 0 1 0 0 1 | 0 0 1 0 0 0 1 0 0 1 0 0 1 0 0 1 | K = 5

-
- Keep growing when the currently used stage hits 50%
-

Copyright © 2020 Raven Computing

Our implementation

- We use our own bit vector implementation. It internally caches the number of set bits
- We use the *Murmur3* (128 bit) algorithm for hashing (may change in the future)
- Adjusted the *Murmur3* Java-implementation from Google's Guava library to perform hashing without allocating heap memory
- No use of *ByteBuffer* objects since endianness does not matter (no cross-language support)
- Only use 64 bits from the 128 bit hash value
- Only compute one real hash value and modify it for every slice as discussed in [3]

Copyright © 2020 Raven Computing

Evaluation

For benchmarking we searched for static Bloom filter implementations on GitHub (in Java)

We ended up including one Cuckoo filter as well to have a more complete view over all publicly available solutions

All conducted test runs executed on an *Intel i7-9700K* CPU

Copyright © 2020 Raven Computing

Evaluation

The following external test subjects were finally included into the benchmarks:

- **Baqend** - Various implementations of static Bloom filters provided by Baqend [4]
- **Google** - Their implementation of a static Bloom filter as provided by the open source Guava library [5]
- **Gunlogson** - An implementation of a Cuckoo filter by Mark Gunlogson [6]
- **Sangupta** - An implementation of a static Bloom filter by Sandeep Gupta [7]

Copyright © 2020 Raven Computing

Evaluation

Comparing filters when given the effective set capacity initially

Copyright © 2020 Raven Computing

Evaluation

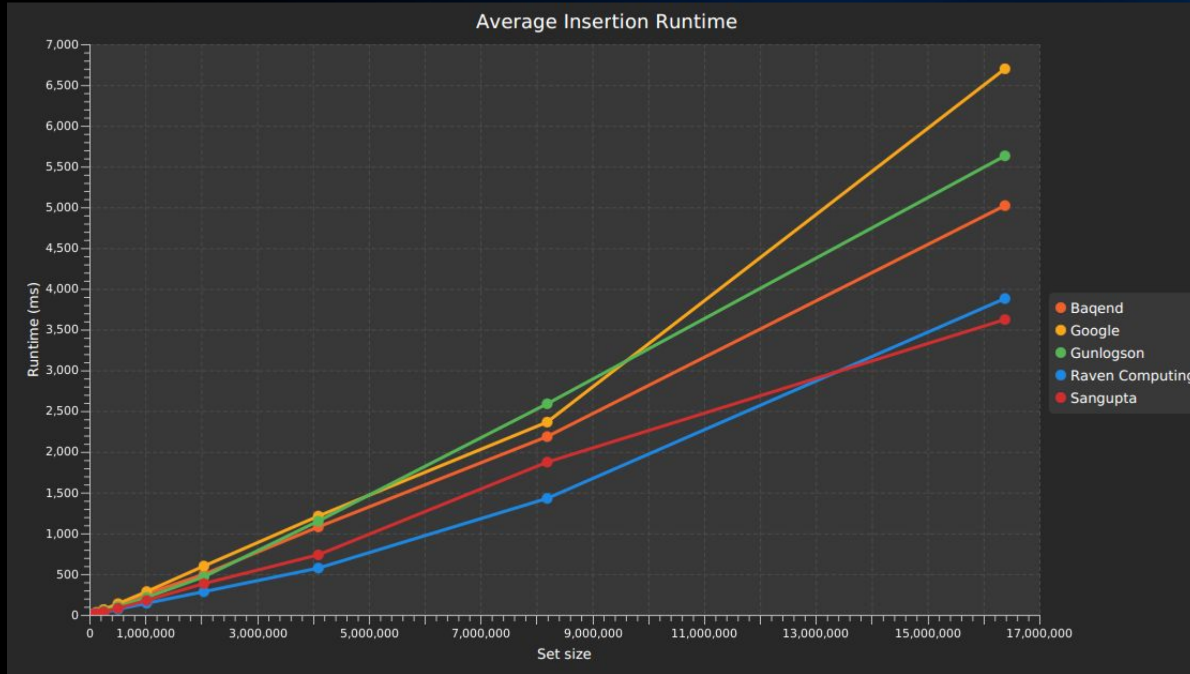


Fig. 1: Average runtime of element insertions for various set sizes and a maximum allowed error probability of 1%

Copyright © 2020 Raven Computing

Evaluation

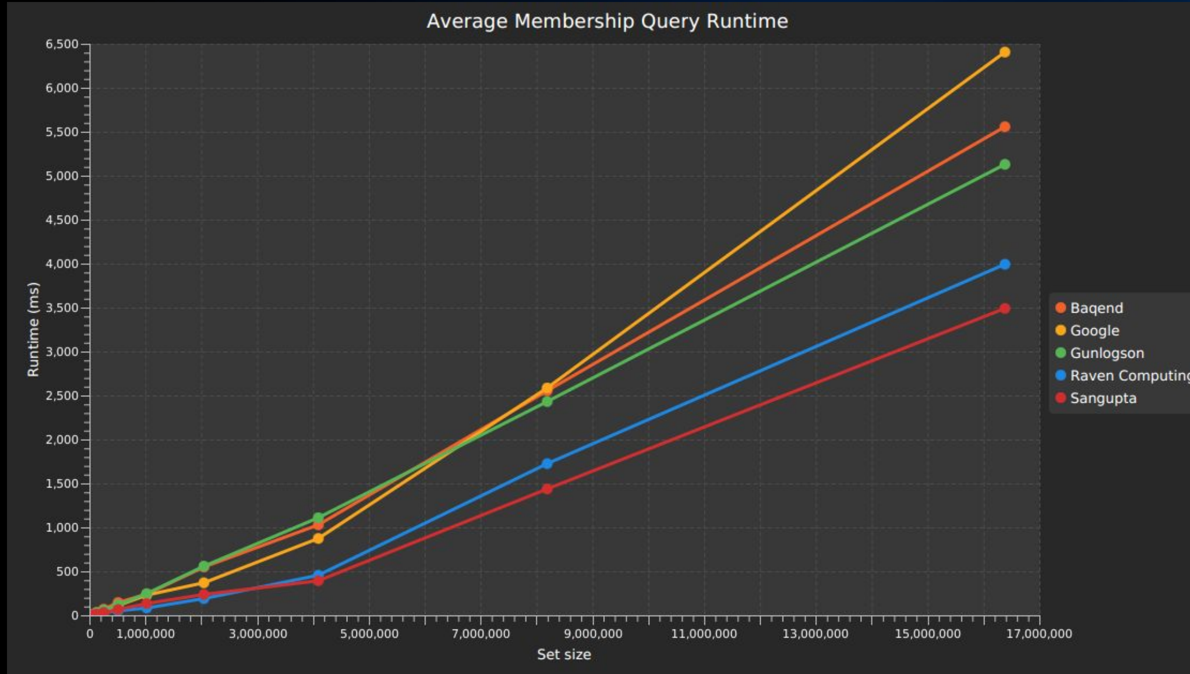


Fig. 2: Average runtime of membership queries for various set sizes and a maximum allowed error probability of 0.1%

Copyright © 2020 Raven Computing

Evaluation

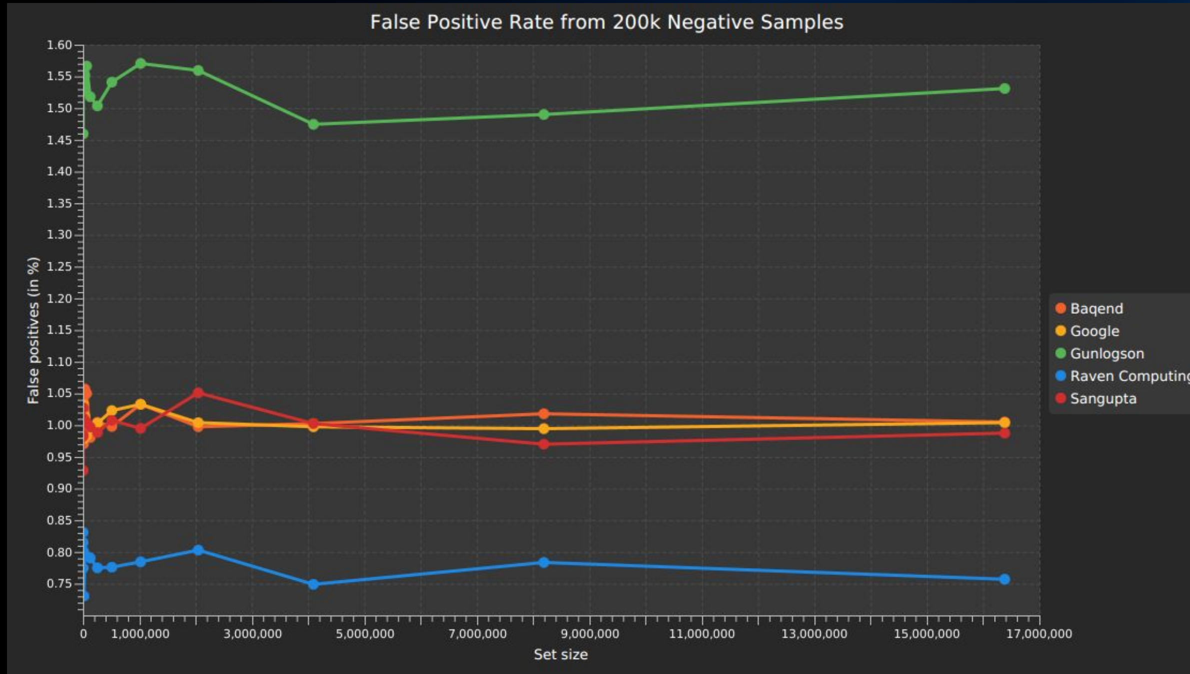


Fig. 3: False positive rate for 200,000 randomly generated negative elements and a maximum allowed error probability of 1%

Copyright © 2020 Raven Computing

Evaluation

Comparing filters when given the effective set capacity initially

- Competitive runtime behaviour compared to static filters
- Outperforms static filters in most cases
- Actual error rate is lower

Copyright © 2020 Raven Computing

Evaluation

Key properties

Copyright © 2020 Raven Computing

Evaluation

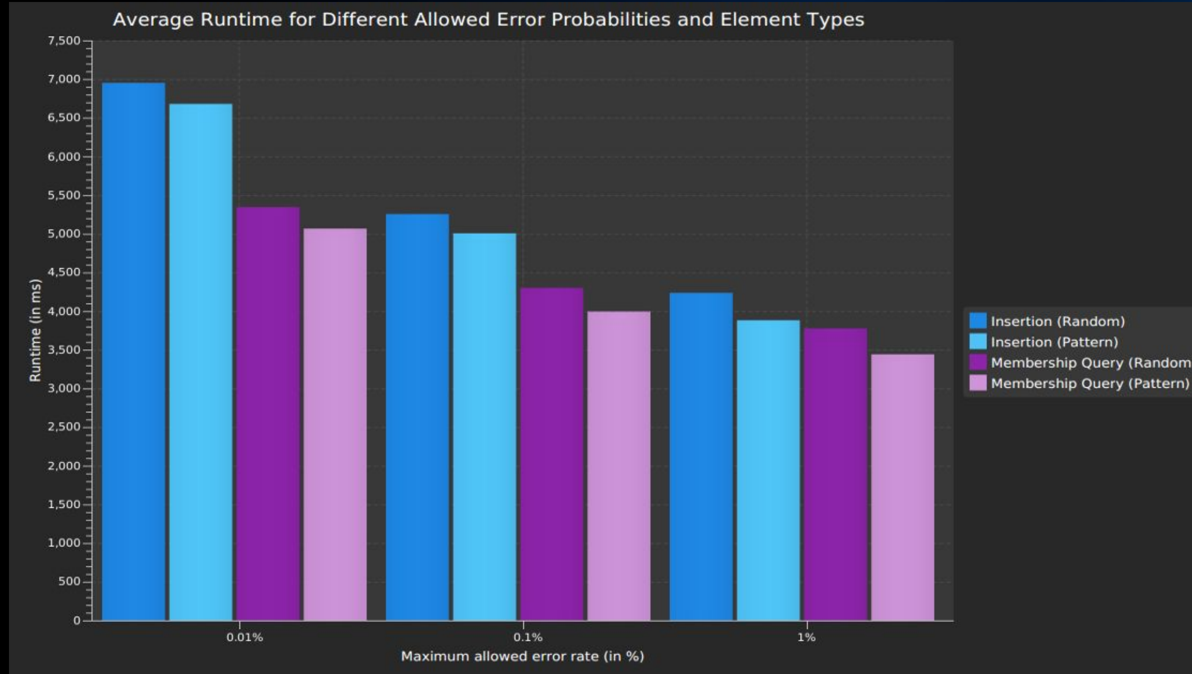


Fig. 4: Average runtime for different maximum allowed false positive rates, element types and filter methods for a set with 16,384,000 elements

Copyright © 2020 Raven Computing

Evaluation

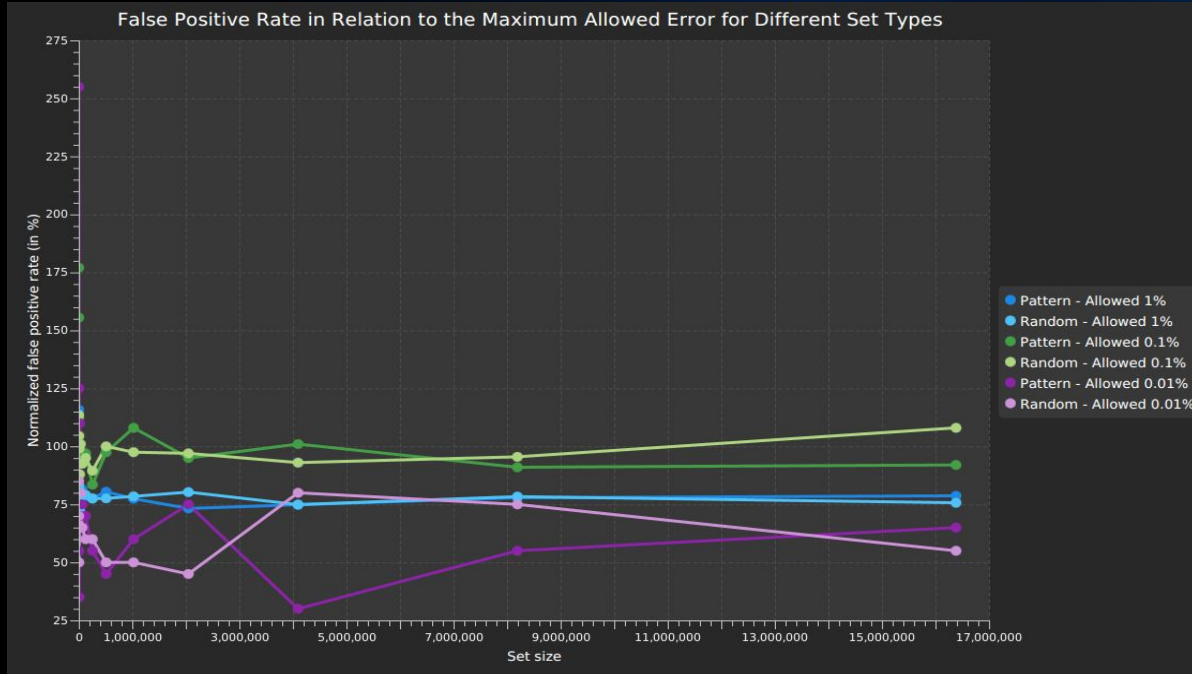


Fig. 5: Normalized false positive rate for different element types and maximum allowed error rates as a function of set size. The 100% mark corresponds to the maximum allowed error

Copyright © 2020 Raven Computing

Evaluation

Key properties

- Membership queries are faster than element insertions
- Demanding a lower maximum error rate makes filter operations slower
- Insignificant differences between random and pattern structured data
- Good adherence to specified maximum error rate

Copyright © 2020 Raven Computing

Evaluation

Comparing filters when our implementation is aggressively scaling

Copyright © 2020 Raven Computing

Evaluation

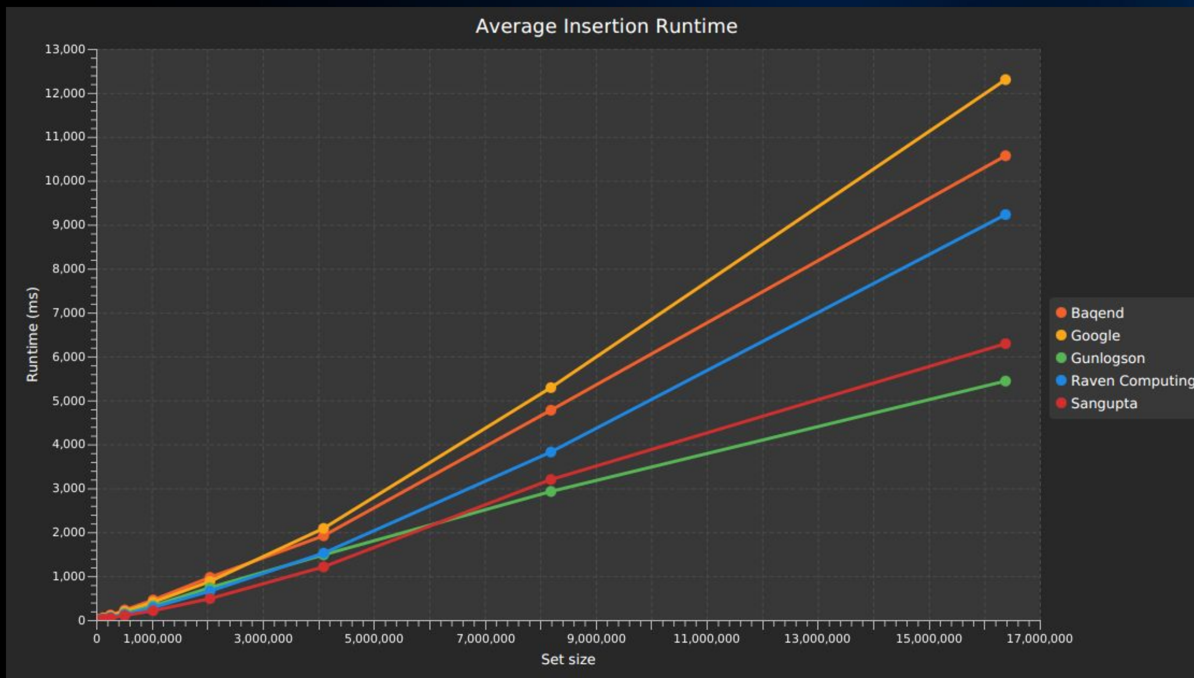


Fig. 6: Average runtime of element insertions for various set sizes and a maximum allowed error probability of 0.01%. Our scaling filter is constructed with an initial capacity of 1,000 elements

Copyright © 2020 Raven Computing

Evaluation

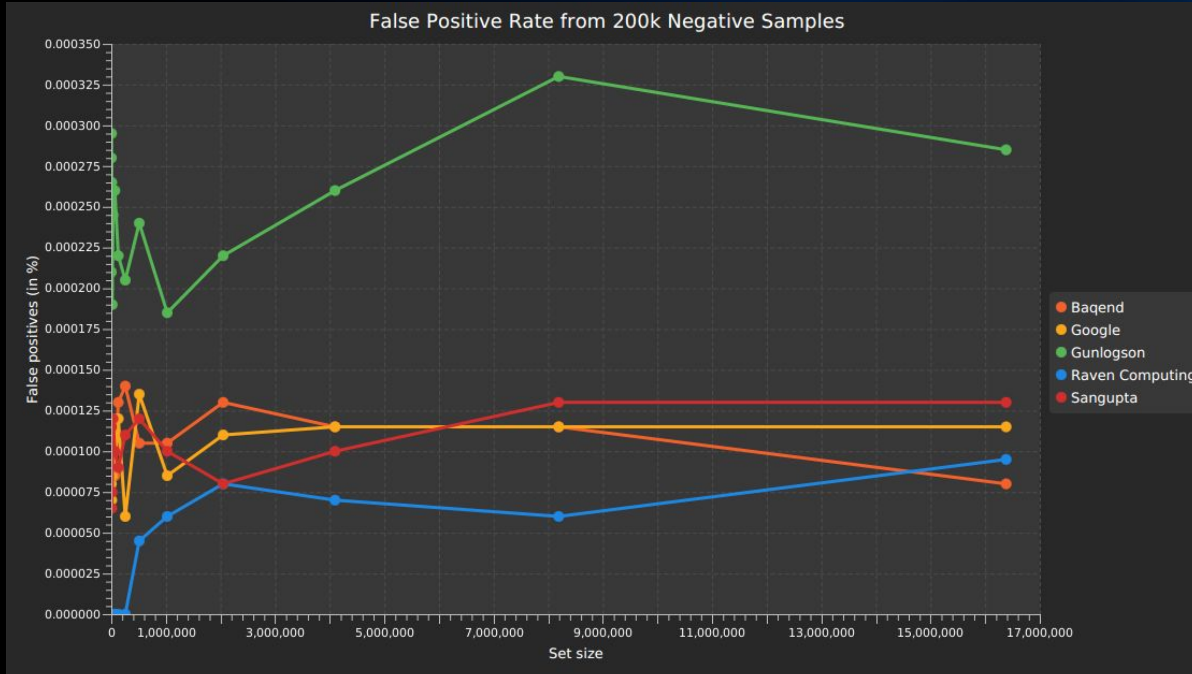


Fig. 7: False positive rate for 200,000 negative samples and a maximum allowed error probability of 0.01%. Our scaling filter is constructed with an initial capacity of 512,000 elements

Copyright © 2020 Raven Computing

Evaluation

Comparing filters when our implementation is aggressively scaling

- Runtime starts to deteriorate slightly after approx. 8 million inserted elements
- Runtime never gets really out of proportion
- But: at a certain point of scaling it starts to introduce a performance penalty
- The actual error rate has more outliers when scaling
- The actual error rate mostly adheres to the specified maximum allowed rate for practically plausible scenarios

Copyright © 2020 Raven Computing

Conclusions

Our implementation provides competitive results compared to other publicly available implementations

Scaling takes a considerable time until it starts to significantly deteriorate filter performance

A scalable Bloom filter is particularly suitable for resource constraint programs

Copyright © 2020 Raven Computing

Where do we go from here?

www.raven-computing.com/research/bitspark

You can download the paper

You can download the code

You can download this presentation

The scalable Bloom filter implementation will be incorporated into the **Claymore** library in version **3.0.0**

Copyright © 2020 Raven Computing

References

[1] B.H. Bloom, Space/time trade-offs in hash coding with allowable errors, Communications of. ACM, vol. 13, no. 7, 1970

[2] P.S. Almeida, C. Baquero, N. Preguica, and D. Hutchison, Scalable Bloom filters, Information Processing Letters, vol. 101, pp. 255–261, 2007

[3] A. Kirsch and M. Mitzenmacher, Less hashing, same performance: building a better bloom filter, European Symposium on Algorithms, pp. 456–467, 2006

[4] Baqend (org.), Library of different Bloom filters in Java (...), GitHub, 2015, Access via https://github.com/Baqend/Orestes-Bloomfilter

[5] Google (org.), Google core libraries for Java , GitHub, 2010, Access via https://github.com/google/guava

[6] M.Gunlogson, High performance Java implementation of a Cuckoo filter, GitHub, 2016, Access via https://github.com/MGunlogson/CuckooFilter4J

[7] S. Gupta, Bloom filters for Java, GitHub, 2014, Access via https://github.com/sangupta/bloomfilter

Copyright © 2020 Raven Computing

Thank you.