# A Scalable Bloom Filter Implementation and Benchmarks in Java

Phil Gaiser
*Raven Computing GmbH*
Bremerhaven, Germany
phil.gaiser@raven-computing.com

***Abstract*—Bloom filters are a widely used data structure as they allow for very efficient implementations regarding space usage and runtime requirements. Over the past decades there have been numerous proposals for adjusting the core idea of Bloom filters in order to allow removal and counting of elements as well as other features. However, many available concepts and implementations do not allow scaling by dynamically changing set sizes. In this paper we propose our implementation of a scalable Bloom filter in the programming language Java based on the theoretical work published by Almeida et al. in 2007. Furthermore, we conduct measurements on how well our filter performs compared to other publicly available (static) implementations. We can show through benchmarks that our implementation yields competitive results with regard to runtime and false positive rates while facilitating dynamic scaling for cases with considerable set growth.**

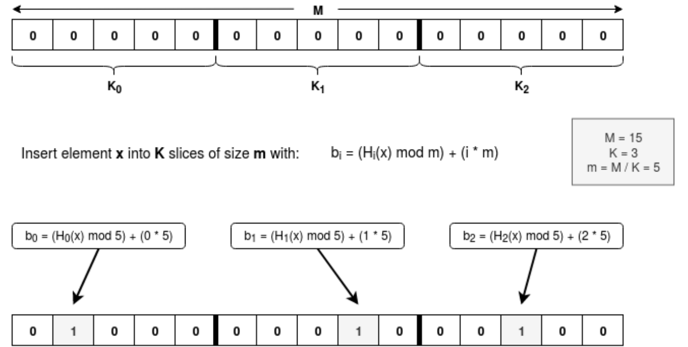


Fig. 1. A basic Bloom filter divided into K slices.

## 1. Introduction

Bloom filters [1] are an important data structure in computer science as they allow the storage of sets without having to keep all elements inserted into the set in memory. Thus they have many practical applications [2] [3] particularly for software environments with limited resources e.g. embedded systems like network routers and smartphones. Bloom filters achieve their outstanding performance by essentially storing hash values of the original element, mapped to indices of a bit vector instead of storing each element directly [1]. In practice, one or more hash functions generate hashes of an object which are then used to set various bits inside a bit vector. Later on, when checking the existence of that object with the same operation, all bits at the searched indices will be set. An inserted element of any arbitrary size is therefore represented only by a few bits inside the filter. This compression of information makes Bloom filters very space efficient as a data structure, however, it introduces a probability of error. That is, a filter might erroneously affirm the presence of a specific element in its set without it ever having been inserted, which is called a false positive. On the other hand, a Bloom filter will never deny the presence of an existent element in its set, i.e. once inserted, subsequent membership queries for that element are guaranteed to be affirmed.

As discussed in [3] [4] we divide each filter into $K$ slices of equal size. This will assure that all elements always set $K$ bits in the entire vector when inserting them into the filter. For example, Fig. 1 illustrates the insertion of a first element into a filter of size $M = 15$ bits with $K = 3$ slices and each slice having $m = 5$ bits. As the vector in the example has now a total of 3 bits set, the fill ratio $p$ is now 20% (3/15 = 1/5). If the output of all hash functions $H_0(x)$, ..., $H_i(x)$ is evenly distributed then as a consequence, the false positive probability $P$ of a subsequent element membership query is:

$$P = p^K = \left(\frac{1}{5}\right)^3 = 0.008 = 0.8\%$$

By inserting more distinct elements into the filter, the fill ratio and therefore the false positive rate will increase accordingly. A Bloom filter is optimally full when half of the bits inside its bit vector are set [4]. Usually a maximum error rate can be specified when a filter is initialized which determines the highest acceptable probability that a membership query of any element results in a false positive.

In the following we give a brief description of related work, after which we go into the most important conceptual traits of the scalable Bloom filter variant in Section 3. We then describe the key points of our implementation in Section 4. The results of our conducted benchmarks are discussed in Section 5. We end by summarizing our findings in Section 6 with our conclusions.

## 2. Related Work

The core concept of Bloom filters as proposed by the original paper has already been extended in numerous ways to add new capabilities and features. For example, the authors in [5] [6] [7] propose variants of a counting Bloom filter which allows removal of elements. In [8] [9] [10] an extension is proposed to introduce subset matching capabilities. As an alternative to Bloom filters, Cuckoo filters [11] [12] also represent

an approximate set data structure while being conceptually based on Cuckoo hash tables. Additionally, implementations of scalable Bloom filters exist for other languages [13] [14].

## 3. Scalable Bloom Filters

Although the original concept of a Bloom filter has been widely adopted, extensions and implementations usually require that the initial capacity is specified when the filter is constructed in memory. Normally this is taken as a constant which means that the filter capacity cannot dynamically change at runtime. While in some situations this does not mark a relevant restriction, in other use cases where the expected amount of elements is not or cannot be known in advance, this is clearly problematic. Even though it is theoretically possible to add more elements to a static filter than the maximum amount specified, it is not practical to do so because the false positive rate increases rapidly as the fill ratio starts to exceed the stochastically optimum. To avoid this problem, the authors in [4] propose a mechanism by which a new empty Bloom filter is added whenever the fill ratio of the currently used filter exceeds 50%. The new enlarged filter is then used for any subsequent element insertions (read/write access) while all previously utilised filters are only used for membership queries (read access). Each new filter is larger and has more slices than its predecessor and is called a stage of the underlying Bloom filter data structure. The number of slices at any given stage is:

$$K_i = \left\lceil K_0 + i \; log_2\left(\frac{1}{r}\right)\right\rceil$$

where $r$ is the tightening ratio of the Bloom filter. As shown in [4], choosing $r$ to be around 0.8 - 0.9 gives an optimal ratio. In order to make the Bloom filter as a whole appropriately adaptable to different growth situations, a slice growth factor $s$ is introduced. This is useful when the set size is expected to increase significantly with respect to the initial capacity. In practice the authors suggest to have $s = 2$ for small expected set growth and $s = 4$ for larger set growth. The size of a vector slice in each growth stage is defined by:

$$m_i = m_0 s^{l-1}$$

where $l$ is the number of stages at any given instance. Because $s$ is a factor of $m_i$, setting $s = 4$ will result in a higher slice growth than $s = 2$.

## 4. Our Implementation

Our implementation was designed to be primarily efficient. As every Bloom filter depends heavily on a low-level data structure called a bit vector (also known as bit array or bit set), we provide our own implementation of such. While it is not specifically developed for the use inside a Bloom filter but rather a more general purpose implementation, its performance metrics have shown to be competitive enough with other implementations provided by the SDK.

### A. Growth

Since a scalable Bloom filter must properly resize itself when certain predefined conditions are met, all core methods (inserting and membership query) require more instructions than a static counterpart. When the filter's fill ratio reaches the maximum acceptable value, a new bit vector must be constructed and initialized which requires the allocation of a new object in memory. Additional copy operations may be needed in order to fit the new filter into the internal array structure. The scalable implementation was developed to only execute the minimal amount of additional instructions needed to perform all necessary steps when resizing. Likewise, additional instructions are also needed when performing a membership query because all created bit vectors must be searched for a particular generated hash value. Although in practice an optimization can be applied for membership queries where the first unset bit (zero bit) encountered in a specific sub-vector causes the rest of that vector to be skipped as the element in question cannot be in that vector. Similarly, the first sub-vector which responds positively to a hash membership request will cause the query as a whole to be affirmed immediately.

### B. Hash Algorithm

One of the crucial parts of any Bloom filter is the underlying hash algorithm used. Although cryptographic hash functions are generally developed to have a more evenly distributed (i.e. more random) output than non-cryptographic hash functions, they are also computationally more expensive [8]. But since one key focus of our implementation has been performance and for the general case it is not possible to know the type of data to be used in a filter at compile time, the faster non-cryptographic hash algorithm *Murmur3* (128 bits) was chosen. It is said to have a good enough bit distribution while being performant from a computational perspective [15]. Interestingly, all implementations for the benchmark discussed in Section 5 use the same or a variation of that algorithm for hashing by default.

### C. Allocation Free Code

The original code implementing the *Murmur3* hash algorithm in Java was taken from Google's open source *Guava* library. However, we found that it had some disadvantages for the purpose of our implementation. On the one hand it internally uses *ByteBuffer* objects to store the function input value and to control the endianness of the raw byte values when computing the hash. This is important for an implementation of a general purpose hash algorithm so that the same input on different platforms yields the exact same hash value. But since cross-language support is not a requirement, the endianness of bytes can be ignored. Additionally, the instantiation of a *ByteBuffer* object can be avoided altogether by directly working with index pointers on the primitive array input value. Furthermore, by only returning the first 64 bits from the 128 bit long hash value, an additional allocation of an array can be avoided as well. Therefore our adjusted

implementation of the *Murmur3* hash algorithm returns a single primitive long value and does not make any allocations on heap memory. Thus both methods of our Bloom filter regarding element insertion and membership query do not allocate any objects on heap. The only exception to this is when inserting an element requires a resizing operation of the filter. Since our Bloom filter grows exponentially, this case is negligible with regard to performance because the vast majority of insertions do not entail a resizing operation.

### D. Vector Access

Lastly, we apply a known technique for reducing the number of required hash functions [16]. Ideally, every slice in each vector would have its own independent hash function. However, that would require an insertion and membership query to compute $K$ separate hash values which in practice would be too expensive. As described above, our adjusted *Murmur3* hash function returns a single 64 bit long value. Since indexed array access in Java is bounded by a signed 32-bit integer, we split the hash value into two integers. Then, as we loop through the slices of a vector, we take only the first hash value as an array index and update it in each iteration by adding the second hash value to it. No special logic is required at this point since there is no arithmetic overflow for primitives in Java. Therefore separating each filter into smaller slices takes only three more arithmetic additions in practice.

## 5. Evaluation

In order to evaluate the proposed implementation we developed a program to create benchmarks for both our as well as other openly available implementations. To clarify the underlying methodology we first elaborate on the criteria used to decide what external implementations to include in our tests.

### A. Criteria

As Bloom filters provide practical advantages for software applications, we were in need of a deployable implementation. Regardless of whether it allows for dynamic set growth, in order for a software library to be deployable in production systems, it must exhibit a certain degree of quality. The following requirements must be met for an external implementation to be included in our benchmark tests:

- Filter type - The project must implement either a static or scalable probabilistic set data structure. The exposed API must allow the specification of both an initial capacity as well as a maximum allowed false positive rate the underlying set adheres to.
- Licensing - It must be open source and available on GitHub. All code must be published under a permissive license which allows commercial usage.
- Documentation - The code and API must be documented properly. As a minimum requirement, critical API methods must have proper javadocs attached to them which clarify their usage.
- Quality - A minimum standard regarding code quality must be present. This criterion is not precisely quantified since the corresponding project is not reviewed in detail. However, obvious deficiencies in code quality like obfuscated code, pointless variable declarations etc., or code which is still in alpha development may cause the project not to be accepted as a test subject.
- Availability - Library artifacts must be accessible via the Maven Central Repository.
- Compatibility - Library artifacts must be compatible to Java 8 (minimum).

Even though today there is a large variety of platforms which provide access to open source libraries and projects, the only point of reference for external implementations included in our benchmarks was GitHub. Projects published on other platforms were not considered.

### B. Test Subjects

The following test subjects were finally included into the benchmarks:

- Baqend - Various implementations of static Bloom filters provided by Baqend [17].
- Google - Their implementation of a static Bloom filter as provided by the open source Guava library [18].
- Gunlogson - An implementation of a Cuckoo filter by Mark Gunlogson [19].
- Sangupta - An implementation of a static Bloom filter by Sandeep Gupta [20].

As shown above, the final set of test subjects contains three static Bloom filter implementations and one Cuckoo filter. Because Cuckoo filters are generally used for the same purposes as Bloom filters, i.e. space efficient approximate membership queries of elements, and we found the mentioned implementation which meets our specified requirements, we decided to include it in our benchmarks in order to have a more complete view over all publicly available solutions, even though Cuckoo filters are conceptually different than Bloom filter.

### C. Benchmark Setup

For creating benchmarks of our implementation we ran our program on a server equipped with an *Intel i7-9700K* CPU and 64 GB of RAM. When executed, the program will run all configured test laps sequentially on a single core. In each test lap a concrete filter implementation is instantiated with the specified lap parameters (initial capacity and maximum allowed error rate). Then a data set of the given size is initialized with either pattern or randomized elements. Additionally a second data set is created which holds elements that are known to not be in the first set. Therefore when performing membership queries the actual false positive rate of the underlying filter can be precisely computed. All elements of the first set are then sequentially added into the concrete filter. The runtime is measured with millisecond precision. After all insertions have completed, the runtime for membership queries is computed. Afterwards, the actual false positive rate is measured by performing membership queries for definite negative samples on the filter and counting the number of positive responses.

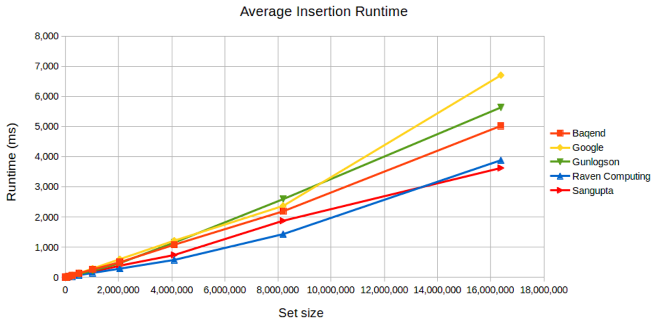


Fig. 2. Average runtime of element insertions for various set sizes and a maximum allowed error probability of 1%.

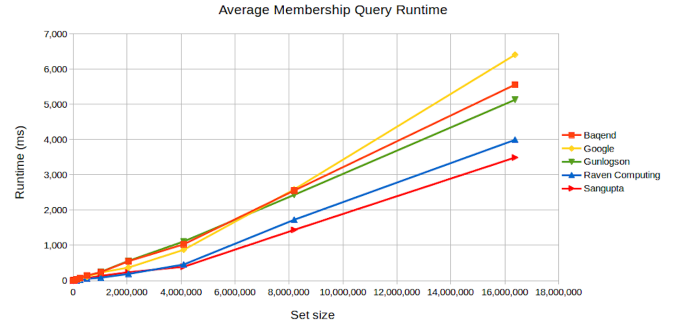


Fig. 3. Average runtime of membership queries for various set sizes and a maximum allowed error probability of 0.1%.

### D. Filter Comparison

This section describes how well our implementation performs with regard to key aspects compared to the other implementations. One important trait of any data structure is runtime behaviour. Since Bloom filters are intended to be used with large sets, it is important to understand how each implementation performs when used with such. In our tests, the set size is increased exponentially in every iteration, starting with 1,000 up to 16,384,000 elements. Fig. 2 shows for each implementation the average runtime for element insertions as a function of set size. As one can see, our filter consequently outperforms all other implementations before being slightly overtaken by *Sangupta* in the last iteration. The absolute differences are, as one might expect, rather negligible for smaller set sizes but increase steadily with larger sets.

The other important runtime behaviour to inspect is that of the membership query method. Fig. 3 shows for each implementation the average runtime for membership queries as a function of set size. The first thing that can be noticed is that in absolute terms, performing membership queries requires less time than inserting elements. This can be explained by the fact that when testing for the presence of an element in a filter, the query can return early on if the first bit tested in a filter is zero. Conversely, an element insertion always has to perform a set operation on all corresponding bits to ensure a correct vector state. Similarly to insertion operations, Fig. 3 shows that our filter mostly outperforms other implementations with the exception of *Sangupta* which takes the lead from around 4m elements and above.

In addition to runtime measurements we conducted tests for determining the actual false positive rate with a set of definite negative samples. That is, we constructed a set with 200,000 elements which are guaranteed to not be in the tested set, then performed membership queries for those elements and counted the number of positive responses (false positives). Fig. 4 shows exemplarily the calculated false positive rate for filters of all implementations as a function of set size. As one can see, our filter is the only one which stays below the requested maximum false positive rate of 1% at all times (cf. Section 5-E). The other implementations mostly reside around the allowed error rate while surpassing it occasionally, with the Cuckoo filter (*Gunlogson*) having an exceptionally high error rate. Evaluations of other tests have shown that the Cuckoo filter implementation does not adhere to the requested error rate in many cases. With regard to our Bloom filter implementation it should be noted though, that its superior performance when it comes to the actual error rate (as shown in Fig. 4) is also due to its scalable capabilities. This should be kept in mind when comparing to other filters as static implementations cannot scale. During a specific test run all filters were given the exact same initialization parameters which means that each filter was constructed to have the same initial capacity. However, our implementation causes the filter to grow when the initial bit vector is filled by more than 50%, which usually occurred for the last inserted elements. Thus when conducting subsequent error measurements our filter has generally more space to perform membership queries against, even though the additional bit vector is poorly used with a fill ratio of usually under 1%. Because of scalability, our filter has therefore a slight advantage compared to the other implementations. When strictly disabling scalable growth, the measured error rate is somewhat closer to that of the other implementations.

### E. Key Properties

This section describes our findings with regard to key properties of our implementation. As the effective runtime behaviour of a data structure depends not only on the input data but also on the initialization parameters, we evaluated the conducted tests to see how the runtime changes with certain data types and parameters. The type of data in this context is defined by being either *random* or *pattern*. As the name suggests, random data is generated randomly during the tests while the other type introduces a certain pattern into the data. This distinction is important because practical deployments of Bloom filters may also work with non-random data, i.e. data which exposes a certain pattern, like file paths in a file system. Since our implementation uses a non-cryptographic hash function internally, a reoccurring pattern in the input data may cause a deteriorated hash quality which in turn might cause a worsened bit distribution in each vector. Combined with other independently changing variables this might result in a great variation of runtime measurements.

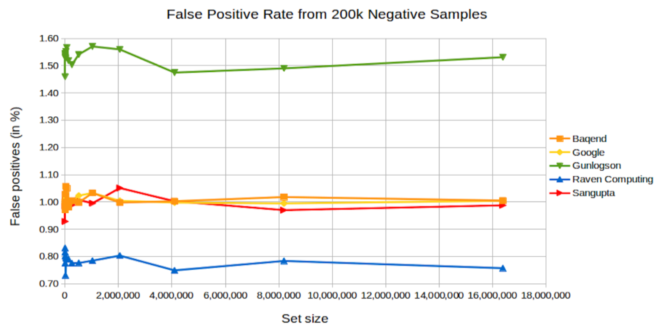


Fig. 4. False positive rate for 200,000 randomly generated negative elements and a maximum allowed error probability of 1%.

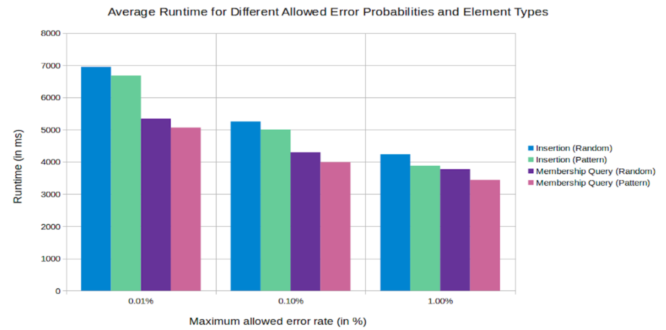


Fig. 5. Average runtime for different maximum allowed false positive rates, element types and filter methods for a set with 16,384,000 elements.

Fig. 5 shows the measured runtime for our filter when working with a set of over 16m elements. It can be seen that in fact there is a difference in effective runtime. First, it can be noted that the data type alone makes only a small difference in runtime. This is also a sign that the underlying hash function used (*Murmur3*) has a random enough hash value output. The observation that randomly generated data has a slower runtime may be due to the fact that randomly generated data is marginally longer than the pattern counterpart and therefore it also takes slightly more time to compute the hash value. Secondly, the measured runtime is also dependent on the specified maximum false positive rate. As the maximum allowed error rate gets tightened, the runtime for both element insertions and membership queries also increases.

Another interesting property is the behaviour of the actual false positive rate. As we have seen in Section 5-D it stays for the most part significantly below the requested maximum rate. But in order to compare the actual rates for different maxima specified, it is better to show actual rates in relation to the corresponding maximum rate, i.e. in percentage terms. Fig. 6 shows that normalized error rate for different data types and specified maximum error rates as a function of set size. It can be seen that for the vast majority of cases, our filter stays below or at the specified maximum error rate with the exception of some outliers. The first observation that can be made is that the maximum specified false positive rate of 0.1% gives the comparably worst results even though it is between the other specified rates of 1% and 0.01% as shown. On the other hand, most actual rates of all configurations adhere to the requested maximum error rate. Notable though, are the few before mentioned outliers which score an exceptionally hight error rate with one measured point having more than double the maximum allowed error. It should also be noted that those exceptional cases were exclusively observed for test runs with pattern structured data. This might indicate a minor deficiency in the output of randomness in our used hash function, as was expected. However, since those exceptional error rates are not occurring in a significant way overall, they are deemed to be irrelevant for practical use cases. It can be said that altogether the actual false positive rate is expected to adhere to the maximum allowed rate in practical applications with insignificant exceptions.

Moreover, testing the scalability of our filter and the effect it has on performance should be elaborated on. As we have shown in the preceding sections, giving all filters the same initial capacity causes our scalable implementation to outperform static filters in most cases (cf. Fig. 2, 3 and 4). Because a scalable filter variant requires more internal operations in order to achieve the scaling capability, it is expected to be slower than its static counterparts when given a fractional initial capacity. We examined this proposition by setting up test laps in which we initialize our filter with a minuscule capacity and then let it aggressively scale up to the highest available set size. Fig. 7 exemplarily shows the insertion runtime for all tested filters where our scalable implementation was given an initial capacity of only 1k elements whereas the static filters had been initialized with the full capacity of the underlying set. Our filter therefore had to scale from 1k to over 16m elements. It is evident that at some point the overhead introduced by scaling outweighs all optimizations as discussed in Section 4. However, Fig. 7 shows that for scaling the filter up to approx. 8m elements, there is no significant deterioration in runtime for element insertions. Indeed, other tests have also indicated that scaling up to moderate set sizes has no measurable effect on runtime behaviour. After a certain point, however, filter performance starts to suffer in a linear manner. For example, when our filter initialized for scaling is supposed to achieve the same performance as when initialized in a static way, an initial capacity of approx. 500k is needed. In other words, giving our filter an initial capacity of about half a million and then scaling up to over 16m elements will result in similar runtime behaviour as when given the full capacity right away. Increasing the set size even further would likely introduce a slight performance penalty.

Lastly, aggressive scaling also has some implications for the actual false positive rate. While under practical conditions scaling does not seem to cause unacceptably excessive error rates, it does, however, somewhat increase the amount of outliers. Generally, the actual error rate approaches the maximum allowed rate when scaling by several orders of magnitude in relation to the initial capacity. For practically plausible deployment scenarios we found that the actual error rate is on average as good or better than the static counterparts.

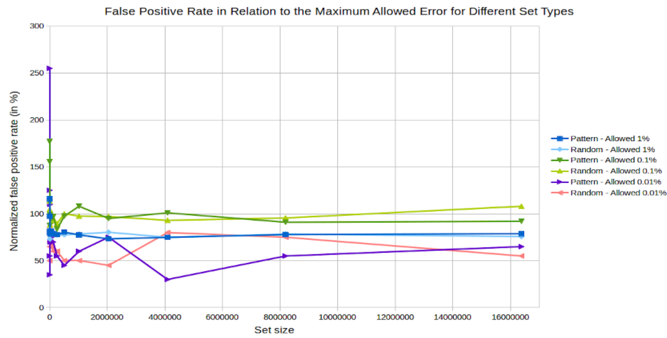


Fig. 6. Normalized false positive rate for different element types and maximum allowed error rates as a function of set size. The 100% mark corresponds to the maximum allowed error.

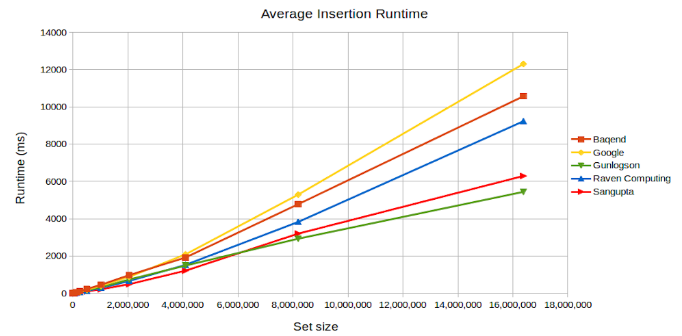


Fig. 7. Average runtime of element insertions for various set sizes and a maximum allowed error probability of 0.01%. Our scaling filter is constructed with an initial capacity of 1,000 elements.

## 6. Conclusions

As we have shown in Section 5, our implementation of a scalable Bloom filter provides competitive results with regard to runtime requirements and maximum allowed false positive rates. We could show through benchmarks that our implementation's performance is not worse than that of publicly available static Bloom filter implementations while also allowing for scalable growth as more elements are inserted into the filter. This capability could therefore be used in Java based programs to reduce memory consumption in cases where the total number of elements the filter is supposed to hold cannot be determined at the time it is constructed. Furthermore, our conducted tests have indicated that the scaling effect also makes the actual error rate adhere in a better way to the maximum desired error rate compared to the discussed static implementations. This overall yields better performance for many practically plausible deployment scenarios. It may be advisable for resource constrained programs to use the proposed scalable implementation to optimize memory usage. It may provide an appropriate API to set specific filter parameters in order to make the implementation more adaptable to different use cases and situations.

## 7. Acknowledgment

The underlying theoretical work for our scalable Bloom filter implementation and the introduction of the theoretical concepts is based on a paper published by Almeida et al. in 2007 [4]. It is referred to here for a more detailed review of the mathematical properties of this variation of Bloom filters.

## References

[1] B.H. Bloom, Space/time trade-offs in hash coding with allowable errors, Communications of. ACM, vol. 13, no. 7, 1970
[2] A. Broder and M. Mitzenmacher, Network Applications of Bloom Filters: A Survey, Internet Mathematics, 2002
[3] F. Chang, W. Feng and K. Li, Approximate caches for packet classification, IEEE INFOCOM, vol. 4, pp. 2196–2207, 2004
[4] P.S. Almeida, C. Baquero, N. Preguica, and D. Hutchison, Scalable Bloom filters, Information Processing Letters, vol. 101, pp. 255–261, 2007.
[5] O. Rottenstreich, Y. Kanizo and I. Keslassy, The variable-increment counting Bloom filter, IEEE/ACM Transactions on Networking, vol. 22, no, 4, pp. 1092–1105, 2013
[6] F. Bonomi, M. Mitzenmacher, R. Panigrahy, S. Singh and G. Varghese, An improved construction for counting bloom filters, European Symposium on Algorithms, pp. 684–695, 2006
[7] L. Li, B. Wang and J. Lan, A variable length counting Bloom filter, 2010 2nd International Conference on Computer Engineering and Technology, vol. 3, pp. V3–504, 2010
[8] A. Pagh, R. Pagh and S.S.Rao, An Optimal Bloom Filter Replacement, Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms, pp. 823–829, 2005
[9] B. Chazelle, J. Kilian, R. Rubinfeld and A. Tal, The Bloomier filter: an efficient data structure for static support lookup tables, Proceedings of the fifteenth annual ACM-SIAM symposium on Discrete algorithms, pp. 30–39, 2004
[10] C.W. Mortensen, R. Pagh and M. Ptraçcu, On dynamic range reporting in one dimension, Proceedings of the thirty-seventh annual ACM symposium on Theory of computing, pp. 104–111, 2005
[11] B. Fan, D.G. Andersen, M. Kaminsky and M.D. Mitzenmacher, Cuckoo filter: Practically better than bloom, Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies, pp. 75–88, 2014
[12] M. Kwon, P. Reviriego and S. Pontarelli, A length-aware cuckoo filter for faster IP lookup, 2016 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), pp. 1071–1072, 2016
[13] Bitly (org.), Scaling, counting, bloom filter library, GitHub, 2013, Access via https://github.com/bitly/dablooms
[14] J. Baird, Scalable Bloom Filter implemented in Python, GitHub, 2013, Access via https://github.com/jaybaird/python-bloomfilter
[15] A. Appleby, MurmurHash, 2008, See https://sites.google.com/site/murmurhash
[16] A. Kirsch and M. Mitzenmacher, Less hashing, same performance: building a better bloom filter, European Symposium on Algorithms, pp. 456–467, 2006
[17] Baqend (org.), Library of different Bloom filters in Java with optional Redis-backing, counting and many hashing options, GitHub, 2015, Access via https://github.com/Baqend/Orestes-Bloomfilter
[18] Google (org.), Google core libraries for Java , GitHub, 2010, Access via https://github.com/google/guava
[19] M. Gunlogson, High performance Java implementation of a Cuckoo filter, GitHub, 2016, Access via https://github.com/MGunlogson/CuckooFilter4J
[20] S. Gupta, Bloom filters for Java, GitHub, 2014, Access via https://github.com/sangupta/bloomfilter