The Gaussian Bloom Filter

Martin Werner and Mirco Schönfeld

Ludwig-Maximilians-Universität in Munich, Germany,
martin.werner@ifi.lmu.de, mirco.schoenfeld@ifi.lmu.de

Abstract. Modern databases tailored to highly distributed, fault tolerant management of information for big data applications exploit a classical data structure for reducing disk and network I/O as well as for managing data distribution: The Bloom filter. This data structure allows to encode small sets of elements, typically the keys in a key-value store, into a small, constant-size data structure. In order to reduce memory consumption, this data structure suffers from false positives which lead to additional I/O operations and are therefore only harmful with respect to performance. With this paper, we propose an extension to the classical Bloom filter construction which facilitates the use of floating point coprocessors and GPUs or additional main memory in order to reduce false positives. The proposed data structure is compatible with the classical construction in the sense that the classical Bloom filter can be extracted in time linear to the size of the data structure and that the Bloom filter is a special case of our construction. We show that the approach provides a relevant gain with respect to the false positive rate. Implementations for Apache Cassandra, C++, and NVIDIA CUDA are given and support the feasibility and results of the approach.

Keywords: Bloom Filter, Database Design, Data Structures

1 Introduction

Nowadays, the Internet has become one of the most important information hubs of our societies. While in the past, the Internet was used mainly as a source of information, it is becoming more and more a hub for user-generated and sensor data. The Internet-of-Things paradigm envisions that more and more devices of daily life transmit information to the Internet and consume information retrieved over the Internet for flexible service delivery.

Due to the wide adoption of large-scale cloud computing approaches in data management and due to the rise of "NoSQL" databases for solving the problems of management of large collections of data in a distributed way with linear scaling, a lot of approaches have been discussed in order to index and manage the high amount of data in a more flexible way. The data flow can be subsumed as a process and the most important aspects for big data applications stem from this data flow which can be enumerated as follows:

1. *Data Collection:* The process of moving the data into a big data infrastructure.

2. *Data Distribution:* The process of distributing the data reasonably to several instances for performance and error-tolerance.
3. *Storage and Retrieval:* The operations used to store and retrieve data from persistent storage.
4. *Searching and Indexing:* The operations making data access flexible and reasonably fast for applications.
5. *Big Data Analytics:* The area of analyzing the meaning of large datasets.
6. *Visualization:* The challenges of making these outcomes visible and understandable for humans.

It is estimated that the amount of data on the Internet doubles roughly every two years and this data is the commodity of information society.

For adressing the first three challenges, a well-known data structure has been used a lot: The Bloom filter. This data structure is able to model the containment relation of variable-sized small sets with high accuracy, constant memory, small computation cost, and some false positives. With respect to big data infrastructures, it has been widely used in order to be able to skip disk reads or network transmission and manage access to information in key-value stores. With respect to data distribution, it can be applied for set reconciliation between two large datasets [5, 4]. With respect to data collection it has been widely applied to network routing and related problems [13, 8].

With this paper, we provide an extension to the classical Bloom filter which can be used in big data applications. The extension is able to use additional memory in order to reduce the false-positive rate of a Bloom filter while the underlying classical Bloom filter can be extracted easily. Additionally, our approach is based on using floating point calculations, which can be shifted to a graphics card or other co-processor exploiting parallel computation capabilities.

We provide an implementation[1] of our approach for the well-known NoSQL database Apache Cassandra and highlight the performance from two perspectives: A moderate increase in running time for key insertion is traded against a higher rate of skipping irrelevant data blocks especially for data blocks for which the number of elements stored is smaller than the expected number of elements.

The remainder of this paper is structured as follows: We first review the needed background on classical Bloom filters in Section 2. Then, we detail our extension to this construction, the "Gaussian Bloom Filter" in Section 3. In Section 4, we evaluate our approach with respect to its false positive rate, the number of bits used for floating point representation, the performance of our implementation for the well-established key-value store Apache Cassandra, and the performance of different variants implemented for the CPU and the GPU in C++. Section 5 concludes the paper.

[1] supplementary material is available at http://trajectorycomputing.com/gaussian-bloom-filter/

2 Bloom Filter

The Bloom filter, introduced by B. H. Bloom in 1970, is a probabilistic data structure for representing small sets of strings in a space-efficient way [3]. A Bloom filter can be used to rapidly check if the underlying set contains a certain element. A Bloom filter's main asset is the absence of false negative filter results: A Bloom filter will never claim that an element is not in the filter while it has been inserted to the filter. But, a Bloom filter does report false positive results stating that the underlying set contains the inquired element although it has not been added. The amount of expectable false positive filter responses depend on the filter's configuration, mainly.

Bloom filters are a central element of several modern distributed systems such as Google BigTable [6], Apache Cassandra [1], and even BitCoin [9]. Their manifold application led to a lot of research investigating the specialization of the basic variant towards specific application scenarios. Up to now, there are counting Bloom filters, compressed Bloom filters, time-decaying Bloom filters, stable Bloom filters, coded Bloom filters and several more [4, 10].

The basic filter itself consists of a binary hash field of fixed size supporting the insertion of and querying for single elements. The following definition gives the details.

Definition 1 (Bloom Filter). *A Bloom filter is given by a binary hash field F of fixed length m. Fix a set h_i of k pairwise independent hash functions mapping the universe U to the set $\underline{m} = \{1, 2, \ldots, m\}$. In this situation, the following set of operations defines a data structure describing sets. The empty set is represented by an all-zero hash field $F = 0$.*

1. *Insert(F, e): Set all bits of F to the value 1, which are indexed by the results of all k hash functions h_i applied to the element e.*
2. *Test(F, e): Return true if and only if all bits in F, which would have been set by the corresponding Insert operation, equal 1.*

From this definition, one can see why a Bloom filter does not allow for false negative filter responses but may report false positives: the test operation uses k hash functions to check for the addressed bits being set to 1 – these are the same k hash values the insert operation would have used to set those bits. So, if there is only one position of the addressed bits $F[x] = 0$ the element could not have been inserted into the filter. On the other hand, all tested bits could have been set to one by the insertion of various other elements due to hash collisions. In this case, the test operation would respond falsely positive.

Since false positive responses often lead to costly operations their occurences should be minimized. But, the probability of false positives can be calculated. Since all k hash functions are assumed to be pairwise independent, the probability of a false positive can be expressed as the probability that all k hash functions hit a one for an inquired element. With some simplifications, this can be approximated via the fraction of zeros in the filter.

The probability for a slot still being unset after evaluating one single hash function is

$$1 - \frac{1}{m}.$$

For the insertion of n different elements hashing is performed kn times since every insertions uses k different hash functions. Therefore, the probability p for a slot being zero after n insertions can be expressed as

$$p \approx \left(1 - \frac{1}{m}\right)^{kn} \approx \exp\left(-\frac{kn}{m}\right).$$

The probability of a false positive $P(fp)$ can now be given as the complementary event to p since a false positive occurs if all k hash functions hit a bucket "not being zero":

$$P(fp) \approx (1-p)^k$$

To obtain the minimal false positive probability the first derivative of $P(fp)$ with respect to k is taken and set to zero resulting in an optimal parameter k_{Opt}:

$$k_{\mathrm{Opt}} = \frac{m}{n} \log 2.$$

This leads to a fraction of zero of

$$p_{\mathrm{Opt}} = \frac{1}{2}$$

and a false-positive probability of

$$P_{\mathrm{Opt}}(fp) = \left(\frac{1}{2}\right)^{\frac{m}{n}\log 2} \approx 0.6185^{\frac{m}{n}}.$$

An optimal Bloom filter configuration leads to roughly half of the bits being zero and half of the bits will be one. This motivates from an information-theoretic perspective that it is impossible to spill in more information into the hash-field: The entropy in this case is maximal. The optimal configuration depends on the number of elements being inserted, the size of the bit field and the number of hash functions being used.

When thinking about the Bloom filter, one quickly realizes that only zero-valued slots contribute information for reducing false-positives: A false-positive is rejected, if at least one of the addressed cells is zero. From this observation, we motivate our extension: We want to introduce more information into each "zero-valued" slot, namely some information about the ordering of hash functions. In this way, the surrounding fields store some information on which hash function has actually set the enclosed bit. This helps to further reduce the occurence of false positives since the test operation is now able to decide if a bit was set by storing the inquired element or a different one and additionally reject some of the false-positives in which the ones are addressed from different hash functions. The following section gives details on this construction.

3 Gaussian Bloom Filter

For the construction of the Gaussian Bloom filter, we first replace all binary slots of the hash field F with small floating point numbers. We further extend the insert and test operation as follows:

Definition 2 (Gaussian Bloom Filter). *A Gaussian Bloom filter is given by a hash field F of fixed length m composed of small cells storing floating point numbers. Fix a set h_i of k pairwise independent hash functions mapping the universe U to the set $\underline{m} = \{1, 2, \ldots, m\}$. In this situation, the following set of operations defines a data structure describing sets. The empty set is represented by an all-zero hash field $F = 0$.*

1. *Insert(F, e): For each of the k hash functions h_i create a Gaussian probability density function*
$$\mathcal{N}_i = \mathcal{N}(h_i(e), i)$$
with mean $\mu = h_i(e)$ given by the hash function value and standard deviation $\sigma = i$ given by the hash function index. With respect to the hash field F, we set all entries to the maximum of the current value in the slot and the value in the normalized signature function
$$\tilde{\mathcal{N}}_i = \frac{\mathcal{N}_i}{\max(\mathcal{N}_i)}$$
which attains its maximal value 1 at $h_i(e)$ and values between 0 and 1 elsewhere:
$$F[t] := \max\left(F[t], \tilde{\mathcal{N}}_1(t), \ldots, \tilde{\mathcal{N}}_k(t)\right)$$

2. *Test(F, e): Create the same normalized signature functions $\tilde{\mathcal{N}}_i$ and combine them into the* element signature S *by selecting the maximal value for each slot.*
$$S[t] = \max\left(\tilde{\mathcal{N}}_1(t), \ldots, \tilde{\mathcal{N}}_k(t)\right)$$
Return true, if and only if $F[i] \geq S[i]$ for all $i \in \underline{m}$.

In this construction, the non-maximal slots contain information about the hash function index of the one causing the value of the slot to have changed. For filters with few elements, this can be a lot of information as a lot of non-maximal slots are available to encode such information. Towards an optimally filled filter, the number of available slots reduces.

Choosing the Gaussian kernel for this approach has several reasons: First of all, any bounded function could have been used which can be parametrized with an additional parameter encoding the index of the hash function. Choosing a function which has a single maximum at the location indexed by the hash functions, however, allows for easily extracting the underlying Bloom filter:

Definition 3 (Underlying Bloom Filter). *Given a Gaussian Bloom filter F_G, the underlying Bloom filter F_B can be retrieved by comparing all slots with the maximal value M of the signature functions:*

$$F_B[i] = (F_G[i] == M)$$

This recovers the classical Bloom filter with identical parameters (size, number of hash functions) which would have been generated by directly using the classical construction. This creates an important compatibility when it comes to network applications: We can use the Gaussian Bloom filter without prescribing its use to other components of a distributed system. Moreover, we can use the Gaussian Bloom filter only as long as it actually helps and the additional memory is available.

One subtlety with this definition of the underlying Bloom filter is given by the fact that the signature functions and the resolution and coding of the floating point numbers must be chosen in a way such that no non-maximal cells are rounded to the maximal value. Therefore, we propose to encode the numbers in the cells of a Gaussian Bloom filter in a special way. However, when using floating point processing units such as GPUs in order to perform the filtering operations, keep in mind that rounding can become a problem introducing a novel type of false-positives for the underlying Bloom filter.

Furthermore, the choice of Gaussian kernels can be motivated as making clear the extension nature of our approach: In the space of distributions, the family of Gaussian kernels with smaller and smaller standard deviation converges to the Dirac distribution which is zero everywhere except at 0.

Finally, there are very good speedups for calculating an approximation to the Gaussian function and the calculation can be localized in the array by calculating the function only in a local neighborhood of several multiples of the standard deviation around the indexed hash cells limiting the number of floating point operations per hash function activation.

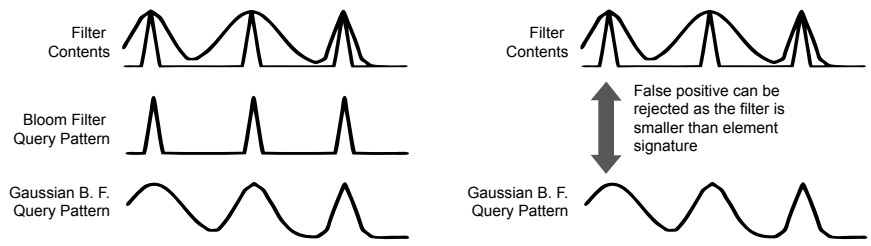
But before we start discussing implementation issues and scalability of the given approach, we first fix some basic results on the filter, its performance and its configuration.

3.1 Properties

In order to discuss the properties of the Gaussian Bloom filter, we start with the following observation that the Gaussian Bloom filter does not allow false negatives:

Lemma 1. *The Gaussian Bloom filter has no false negatives.*

Proof. At any point in time the filter structure is larger than the largest element signature inserted. As the Test operation is based on comparing the signature using greater than or equal to the filter, the query pattern can not be larger than the filter in any slot unless the element has not been inserted to the filter.



(a) Gaussian Bloom Filter Example (b) The Gaussian filter rejects this case

Fig. 1. Gaussian Bloom Filter rejecting a Bloom filter false positive.

The Gaussian Bloom filter allows for false positives just as a Bloom filter does. However, a Gaussian Bloom filter false-positive implies a false positive for the associated underlying Bloom filter.

Lemma 2. *The false positive rate of a Gaussian Bloom filter is smaller than or equal to the false positive rate of a Bloom filter.*

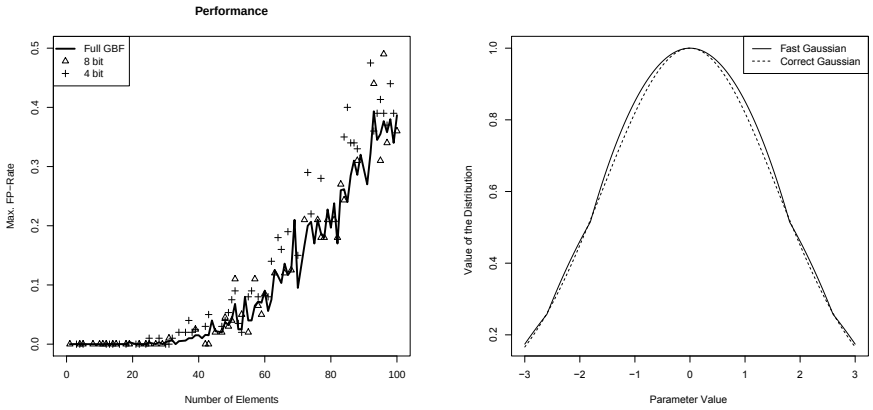
Proof. In the situation of a Gaussian Bloom filter false positive, all slots of the filter are larger than the element signature pattern. This is especially true for the slots, where this pattern attains its maximal value one. However, these slots have been addressed by hash functions of other elements directly.

The following gives a concrete example of a situation in which a Bloom filter reports a false-positive while the Gaussian Bloom filter is able to reject this case.

Example 1. Consider the following situation: Assume the element e_1 with associated hash values $h_1(e_1) = 50$, $h_2(e_1) = 10$, and $h_3(e_1) = 30$ has been added to an empty filter. Assume further that the query element e_2 has hash values $h_1(e_2) = 50$, $h_2(e_2) = 30$, and $h_3(e_2) = 10$

This situation is depicted in Figure 1 on the left: Since all three indexed hash addresses have been set, a Bloom filter returns true for element e_2. A Gaussian Bloom filter on the other hand returns false for the query element. It is able to distinguish the hash functions that addressed the underlying filter index. The right hand side of the figure shows how the query pattern differs from the filter content. Namely, the encoded Gaussian functions for hash addresses 10 and 30 differ in their standard deviation. Therefore, the relevant bits must have been set by different hash functions and, consequently, come from different elements.

Finally, we have to discuss how a Gaussian Bloom filter can be configured. A central element of the analysis of the Bloom filter was the assumption of independence of each bit of information used from other bits. However, in the case of a Gaussian Bloom filter this independence does not exist anymore as different hash values are used when rejecting false positives. Therefore, we are left with the fact that the Gaussian Bloom filter is not worse than the Bloom filter and have to use the Bloom filter analysis in order to choose the optimal configuration in terms of size, number of hash functions, and number of elements.



(a) $m = 256$, $n \in \{1 \dots 100\}$, $k = 6$ (b) Accuracy of the Fast Gaussian

Fig. 2. False-positive rate for small floating points and the accuracy of the fast Gaussian.

3.2 Efficient Representation with Small Counters

Depending on the application domain and the way in which the floating point calculations are performed in practice, it can become quite questionable, whether the amount of memory for example by utilizing single or double precision floating point numbers is reasonable. Furthermore, it is unclear how to prevent rounding up to the maximal value effectively. However, we provide a bit coding tailored to the situation in which even small sizes of 4 bit per slot reach a similar false-positive rate as compared to using an implementation based on an array of `double` values.

In order to encode the floating point values efficiently into a small bit field with t entries, we observe that only values between 0 and 1 need to be modelled. Furthermore, we observe that the maximal value 1 definitely needs a unique bit representation. We chose to model this maximal value by an array of t ones. If the value is not maximal, then we are left with $2^t - 1$ bit combinations which we can use. If we multiply the value with the following scaling factor, then usual rounding will be as expected:

$$\alpha = \frac{2^t - 1}{2^t}$$

In order to obtain the final bit pattern representing the value, let each bit model a fraction of two, the first bit models $\frac{1}{2}$, the second bit $\frac{1}{4}$, and so on. When decoding from this representation, we first check for the distinguished pattern for the maximal value, decode and rescale with α^{-1}.

In order to evaluate the efficiency of the representation, three different filters with equal random sets of elements of varying size are created and then queried with 100 random elements, which have not been inserted into the filters. Figure 2(a) depicts the result of comparing the false-positive rate of a filter based on

full-sized double values as provided by the CPU compared to four bit and eight bit representation. You can clearly see that there is only a small quality loss in comparison between CPU double (bold line) and our binary representations. This is due to the fact that our binary representation uses the bits more efficiently by exploiting the domain limitation to modeling only the interval $[0, 1]$. As to be expected, increasing the number of bits from 4 to 8 bit makes a difference: in general, the larger bit slots tend to reject more false positives.

3.3 Efficient and Local Evaluation of Gaussian Function

For inserting an element to the Gaussian Bloom filter, we have defined to use a Gaussian distribution with two variable parameters. In order to sucessfully apply this in a big data environment with possibly large Bloom filters, we have to optimize the situation into two directions: First of all, it is infeasible to evaluate this function for each cell of the filter and, secondly, it is infeasible to evaluate the correct formula for the Gaussian function containing the exponential function.

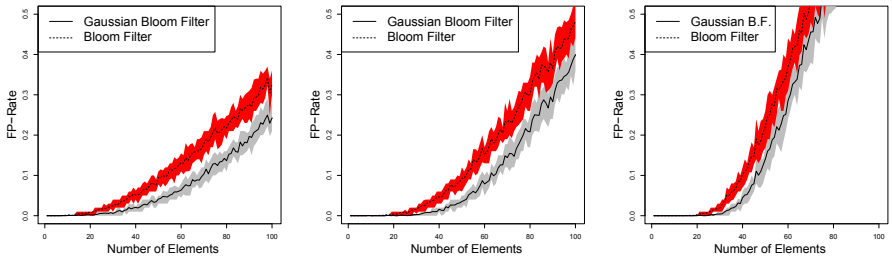
In order to be able to evaluate the Gaussian function for a smaller number of slots, we observe that the tails of the Gaussian function attain zero after rounding quickly. We employ the "3σ-rule" which states that evaluating operations involving the Gaussian functions can be skipped after three sigma due to neglectible size of the function. Outside the interval $[\mu - 3\sigma, \mu + 3\sigma]$ the Gaussian function is smaller than 0.0045, which is small enough to be neglected.

This step, in summary, makes the operations Insert and Test independent from the filter size. They, then, only depend on the number of hash functions used: For each hash function, the number of slots is determined by its index as 3σ determines the amount of slots being accessed. Therefore, the overall number of hash field operations scales with the number of hash functions. Note that even further reduction is possible at the cost of loosing some additional information by imposing a maximal allowed standard deviation. This is especially useful when very high standard deviations dominate too many cells of the filter removing useful ordering information.

In order to be able to create the Gaussian Bloom filter quickly, we adopt a well-known approximation to the exponential function consisting of a integer multiplication, an addition and a binary shift. This method is due to Schraudolph [11]. The speedup of this approximation is astonishing: In our experiments, this implementation was about 30 times faster than the library routine of Java. Figure 2(b) depicts the Gaussian function as computed using the library exponential function compared to the Gaussian implemented using the fast exponential function for the values from the interval $[-3\sigma, 3\sigma]$ in which we actually use the function. Note that outside this interval, the approximation quickly degrades.

4 Evaluation

This section evaluates our new data structure for filtering small sets using an extension to classical Bloom filters with respect to the false-positive rate and in



(a) $k = 3$ (b) $k = 5$ (c) $k = 10$

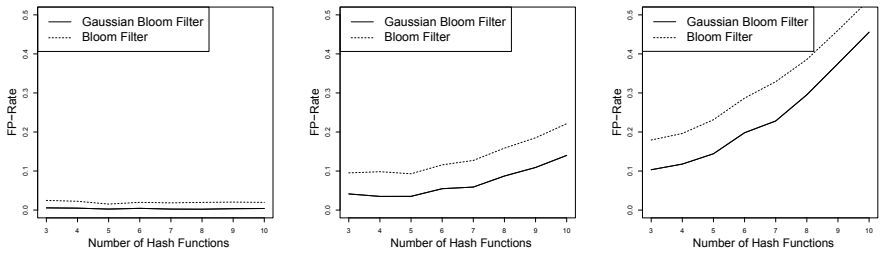
Fig. 3. Comparison of Bloom Filter and Gaussian Bloom Filter Performance ($m = 256$, $k \in \{3, 5, 10\}$ and $n \in \{1, \dots 100\}$).

comparison to the underlying Bloom filter. We performed a lot of experiments with rather high false-positive rates by using small filters and medium numbers of hash functions. These situations serve as example for larger filters with much more elements to insert. In these experiments, a set of random strings has been generated and either SHA-1 [7] or the Murmur Hash [2] have been used to generate hash data. The different hash functions were generated using a different prefix for the hash argument.

Figure 3 compares the Gaussian Bloom filter with an equivalent Bloom filter. In all these figures, the experiments were performed with random string sets and varying parameters many times in order to generate the median performance as the median of the instances measured false-positive rate. Additionally, the first and third quartile area is shaded for both cases. In Figure 3 we see that the Gaussian Bloom filter clearly outperforms the Bloom filter for filters, which have enough zeroes. In the left figure, 256 slots were used together with 3 hash functions. The two figures to the right consider the same situation with an increasing number of hash functions ($k = 5, k = 10$) which leads to more ones in the filter and thereby less possibility to encode ordering information into non-maximal slots. This results in the overall increasing false-positive rate for both filters and the effect that the gain of the Gaussian Bloom filter relative to the Bloom filter gets smaller.

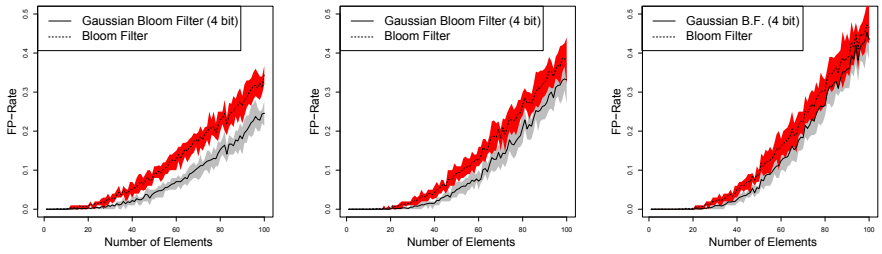
As the gain for the Gaussian Bloom filter construction is larger for underfull filter, we evaluate the false positive rate of a filter with respect to the number of hash functions. Figure 4 depicts the false-positive rate for several situations with different numbers of hash functions. We clearly see that the false-positive rate of the Gaussian Bloom filter is smaller for the number of hash functions and scales similar with the number of hash functions as compared to the Bloom filter.

In summary, this supports the recommendation of using the standard Bloom filter configuration for a Gaussian Bloom filter. If needed, one can also use a slightly smaller number of hash functions for the Gaussian Bloom filter in order



(a) $n = 30$ (b) $n = 50$ (c) $n = 70$

Fig. 4. The effect of the number of hash functions ($m = 256$, $n \in \{30, 50, 70\}$ and $k \in \{3, \ldots 10\}$).



(a) $k = 3$ (b) $k = 4$ (c) $k = 5$

Fig. 5. Performance of a 4-bit Gaussian Bloom Filter ($m = 256$, $n \in \{1 \ldots 100\}$ and $k \in \{3, 4, 5\}$).

to reduce the amount of consumed hash bits. This is epecially important, if hashing of the elements is time consuming, for example in file synchronization applications.

As a third aspect, we have to discuss the performance of the Bloom filter with 4 bit floating point representations. We have already seen in Figure 2(a) that the false-positive rate keeps comparable to a double-based filter for a fixed number of hash functions. Figure 5 depicts the scaling behavior with respect to the number of hash functions. This figure makes clear that the gain of the construction is more sensitive to the number of hash functions as compared to the double approach as the choice of using the hash index as the standard deviation leads to the inclusion of equal non-maximal values in many non-maximal slots (e.g., the largest non-maximal value of the representation) which reduces the overall discriminative gain of the Gaussian signature function. It can be seen that for $k = 3$ the positive effect is clearly visible, but it degrades more quickly for increasing $k \in \{4, 5\}$ as compared to the implementation using double values depicted in Figure 3. In general, the smaller the number of bits in the floating point

representation, the fewer false positives can be rejected. However, the system is keeping between the bounds given by a binary, classical Bloom filter and a filter based on an error-free representation.

4.1 Gaussian Bloom Filter in Apache Cassandra

In order to test our approach in a real-world environment, we decided to extend the open source Apache Cassandra database [1]. Apache Cassandra is based on the Hadoop distributed file system (HDFS) [12] and has been created at Facebook in order to tackle the inbox search problem. Up to the middle of 2011, Facebook was using it for this task. Meanwhile, other large and widely recognized Internet services started using Cassandra inside their backend systems for several tasks including Twitter, Digg, Reddit and others.

From a technological perspective, Cassandra starts with the block-based Hadoop distributed file system and manages a key-value store with which values can be put into the database and retrieved by giving a key. This data is then organized into so-called SSTables and Bloom filters are used to keep track, whether specific keys could be inside specific SSTables in order to reduce I/O. Therefore, each time a key is inserted into an SSTable, the associated Bloom filter is updated. We replace the classical Bloom filter of Cassandra with our modified version and still keep track of the behavior of the original system by managing a classical filter as an instance variable of our new filter. Therefore, performance information from the stress test can not be used to compare the filters, as they contain the overhead of calculating each operation for both filters. Cassandra uses a sophisticated Bloom filter subsystem in which filters are dynamically reconfigured to match the amount of information that needs to be indexed.

We performed experiments on a single-node cluster running our modified version of Cassandra based on Version 2.0.4. We added a new mixed filter containing the Gaussian Bloom filter as well as a copy of the original Bloom filter implementation. Furthermore, logging code has been added to track the performance of each individual Bloom filter operation. For Cassandra, the Bloom filter false positive rate can be configured and we set it to 10% for the experiments in order to have many cases in which both filters will fail in order to reliably count the number of cases in which the Gaussian filter outperformed the classical filter. We logged for a complete run of the default Cassandra stress test the number of situations in which either of both filters was able to reject a false positive with the expected outcome that the proposed Gaussian Bloom filter have no additional false positives and rejects several false positives which kept undetected by the classical Bloom filter. The stress test consists of first writing one million keys into the database, then recalculating all filters using `nodetool scrub` and finally reading one million random keys back from the database.

During this evaluation on a single-node cluster, 23 Bloom filter were generated out of which nine were used in order to manage the stress data. The remaining filters were managing organizational key spaces and registered only few operations. Table 1 collects the number of situations in which both filters

Filter	Bloom SSTable Reads	Gaussian SSTable Reads	Fraction
1	263,937	141,598	0.54
2	625,879	524,120	0.84
3	639,631	539,161	0.84
4	1,230,417	1,163,531	0.94
5	440,584	336,882	0.76
6	271,404	146,835	0.54
7	255,446	138,246	0.54
8	305,535	172,626	0.56
9	260,190	138,611	0.53
Total	4,293,023	3,301,610	0.67

Table 1. Numbers of SSTable reads for each filter during the stress test

were unable to skip an SSTable read. This includes positives as well as false positives.

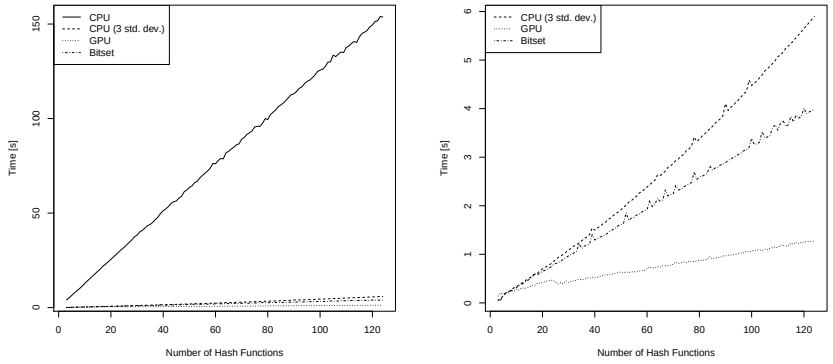
In summary, our novel data structure of a Gaussian Bloom filter was able to skip 991,413 additional SSTable reads as compared to the equally configured Bloom filter, an overall gain of 33% on average. Furthermore, one observes that the reconstruction of Bloom filters in the middle of the experiment before starting reading leads to similar gains of nearly 50%. That is, when the database is restructured and no new data arrives, our structure – with the default configuration of Cassandra – outperforms the given data structure by rejecting 50% of the SSTable reads not rejected by the classical Bloom filter.

The Bloom filter operation performance was also monitored by aggregating the running time of the Insert and Test operation for both, the original Bloom filter and the modified Gaussian Bloom filter. As is to be expected, the Gaussian Bloom filter needs more computations, but still moderately. For the stress test, we observed an increase in running time for the Insert operation by 42%. This is due to the fact that the classical Bloom filter does only access the indexed bits while the Gaussian Bloom filter has to access many more slots, especially when many hash functions are in use such that large standard deviations occur. For the Test operation, however, the measured overhead was only 8.82%. This is due to the fact that testing can be stopped as soon as a single contradiction occurs. This makes our filtering structure more sensible for database applications in which the read-write ratio tends clearly to more reads than writes.

4.2 Performance of a GPU implementation

In order to further motivate the use of Gaussian Bloom filters, we completed another implementation of Gaussian Bloom filters in C++, both for the CPU as well as supported by the GPU via the NVIDIA CUDA programming model.

In this situation, the additional memory of the Gaussian Bloom filter allows for full parallel access without any synchronization and therefore allows for boosting the performance of the Gaussian Bloom filter even more. For the GPU case, we create a thread for each hash function index and calculate the Murmur



(a) (b)

Fig. 6. Performance of various C++ implementations inserting 1,000 random strings.

hash inside the GPU. When this finishes, we create one thread for each slot of the Bloom filter and fill in the original exponential function signature into each slot. Note that this opens up the Gaussian Bloom filter construction to non-local kernels, for which all slots have to be calculated and no optimization comparable to the three sigma rule is available.

Figure 6 depicts the performance of this approach. It shows the time in seconds which is used to insert thousand random strings into a filter. The implementations are pure CPU with and without three sigma rule, the GPU variant, and a bit-packing filter. Note that all implementations except the CPU version without three sigma rule are reasonably fast and scale linearly with the amount of hashing. The small peaks show situations in which the Murmur hash was more complex based on the data. It is interesting that this peak can be observed in all implementations while it is much smaller for the GPU. This is due to the fact the even the calculation of the d hash functions is performed in parallel on the GPU.

The GPU implementation results in several welcome facts: Firstly, the additional memory overhead is not taken from system memory, instead, system memory consumption is reduced as the filter memory is on the GPU. Secondly, the CPU is free for other operations while the hashing and calculation is deferred to the GPU and, finally, exploiting the strong parallelism of typical GPUs allows for calculating kernels globally over the complete hash field and opens up the construction for more complicated kernels.

In order to show the reduction of CPU demand, we performed an experiment on a desktop computer (Intel Xeon E5620 @ 2.4 GHz, 2 CPUs, NVIDIA Quadro 4000, 256 CUDA cores, Windows 7) as follows: Eight threads are started performing a typical workload of randomly generating 10^{12} integer numbers, sorting them and reversing their order in a best effort manner while another thread is

performing 400 insertions into a Bloom filter per second using 50 hash functions and 10,000 slots. This resulted in $1.18 \cdot 10^{18}$ integer workloads completed after 10 seconds for the CUDA implementation compared to $1.15 \cdot 10^{18}$ for the Gaussian Bloom filter on the CPU and $1.14 \cdot 10^{18}$ operations for the binary Bloom filter.

In summary, this approach allowed to free up CPU performance and system memory for applications while maintaining Gaussian Bloom filter in the graphics card.

5 Conclusion

With this paper, we have proposed a novel approach to tuning the classical Bloom filter construction. While the classical Bloom filter performance is optimal when configured with the optimal values, a lot of real-world applications either have to recalculate Bloom filters often or make use of Bloom filters that are configured for many more elements than they actually contain most of the time. In this situation, we propose to model additional information into the Bloom filter data structure into the zero slots. This approach is an extension to the Bloom filter in the sense that maximal values still recover the Bloom filter of the same number of slots, the same hash functions and the same elements. However, we make use of small floating point slots, which can encode from which index of a hash function their value was most influenced. We show in theory and practice that this Bloom filter variant can be used sucessfully. Even for the modern database system Cassandra, which contains sophisticated techniques to optimally configure Bloom filters during operation, a clear gain was visible.

One drawback of this construction is the additional space used to maintain the small floating point numbers. Still, this memory can be on a coprocessing unit such as a GPU freeing up primary memory and CPU capacity for applications. Still, for storage-only backend systems, the additional CPU time and the amount of RAM did not negatively impact the performance of the standard Cassandra stress test of inserting and reading a million keys from a table. We suspect that in many storage backends based on Apache Cassandra the additional amount of memory and CPU is well-invested when the number of complete recalculations of the Bloom filters can be reduced. This is an important direction for future work which, however, can only be performed sensibly with real workloads for the cluster and with a distributed cluster deployment.

The most important advantage consists, however, of the fact that unlike other approaches to provide more efficient filter structures for small sets, our construction is compatible with the original structure and therefore with communication protocols as the underlying Bloom filter can be extracted easily.

References

1. The Apache Cassandra database (2014), http://cassandra.apache.org/
2. Appleby, A.: Murmurhash (2009), http://code.google.com/p/smhasher/
3. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Communications of the ACM 13(7), 422–426 (1970)
4. Broder, A., Mitzenmacher, M.: Network applications of bloom filters: A survey. Internet mathematics 1(4), 485–509 (2004)
5. Byers, J., Considine, J., Mitzenmacher, M.: Fast approximate reconciliation of set differences. Tech. rep., Boston University Computer Science Department (2002)
6. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems (TOCS) 26(2), 4 (2008)
7. Eastlake, D., Jones, P.: US Secure Hash Algorithm 1 (SHA1) (2001)
8. Feng, W.c., Kandlur, D.D., Saha, D., Shin, K.G.: Stochastic fair blue: A queue management algorithm for enforcing fairness. In: INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE. vol. 3, pp. 1520–1529. IEEE (2001)
9. Nakamoto, S.: Bitcoin: A peer-to-peer electronic cash system (2008)
10. Schönfeld, M., Werner, M.: Node wake-up via ovsf-coded bloom filters in wireless sensor networks. In: Ad Hoc Networks, pp. 119–134. Springer (2014)
11. Schraudolph, N.N.: A fast, compact approximation of the exponential function. Neural Computation 11(4), 853–862 (1999)
12. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The hadoop distributed file system. In: 26th IEEE Symposium on Mass Storage Systems and Technologies (MSST). pp. 1–10 (2010)
13. Whitaker, A., Wetherall, D.: Forwarding without loops in icarus. In: Open Architectures and Network Programming Proceedings, 2002 IEEE. pp. 63–75. IEEE (2002)