Modified Bloom filter for high performance hybrid NoSQL systems

A.B.Vavrenyuk, N.P.Vasilyev, V.V.Makarov, K.A.Matyukhin, M.M.Rovnyagin, A.A.Skitev
National Research Nuclear University MEPhI (Moscow Engineering Physics Institute), Russia

Abstract. This article addresses problems of implementation of a modified Bloom filter as an additional module for mass data storage systems in supercomputers with hybrid CPU/GPU architecture. It is proposed to use a modified filter with counters, which makes it possible to monitor not only data addition, but also data removal. A comparative analysis has been made of a serial CPU version and a hybrid version of the Bloom filter based on use of NVIDIA CUDA technology.
[Vavrenyuk A.B., Vasilyev N.P., Makarov V.V., Matyukhin K.A., Rovnyagin M.M., Skitev A.A. **Modified Bloom filter for high performance hybrid NoSQL systems.** *Life Sci J* 2014;11(7s):457-461] (ISSN:1097-8135). http://www.lifesciencesite.com. 98

Keywords: Bloom filter, CPU/GPU hybrid architecture, NVIDIA CUDA, hashing

Introduction

Currently, in many areas of human activities, a rapid growth in the volume of information is observed, which fact lays very high demands to systems of storage, data transmission and processing. Illustrative examples are global physical experiments, in particular, LHC - Large Hadron Collider. According to official data, a distributed grid system of LHC project is focused on storing, distribution and analysis of about 25 petabytes of data (i.e., 25 million gigabytes) [1].

For storing mass data (hundreds of terabytes to tens of petabytes), specialized storages of various architectures are created. To accelerate access to data stored in such systems, developers often turn away from the canons of building classic SQL systems [2]. For the sake of performance, database management is decentralized, and some checks of new data are omitted. As a result of such actions, among other issues, the probability of data search errors increases. Furthermore, a significant number of search queries may be addressed to the data missing in the storage, while time will still be spent for processing these queries. Due to large volume of information, total processing time can become considerable.

Thus, the problem of deleting a search query for not stored data is urgent, and the higher the performance of filtering "needless" queries is, the higher the overall performance of the storage will be.

In order to alleviate the problem of suppressing deliberately needless search queries, in 1970 Burton Bloom proposed a probabilistic data structure, the Bloom filter (BF) [3]. BF displays the information about added elements in an arbitrary data storage that contains information in the "key-value" format. In its classical implementation, the filter is an add-on to a data storage in form of a bit array with length M (Bloom vector, BV) initially populated with zeros. When a new item is added to the storage, K hash functions (HF) are calculated from the key, and the values of HF should range between zero and M-1. BV bits that are determined by results of hash functions are populated with unities.

Data search in the storage is also performed using the Bloom filter. Before addressing the storage directly by the key, K hash functions are calculated for this key and the presence of unities in calculated positions is checked. Presence of at least one zero in either of BV found bits clearly indicates data is absent in the storage. When all calculated BV positions are populated with unities, there are two possibilities. The first is that the data is actually present, and the search operation in the storage will be successful. The second possibility is that the information in the data storage is missing, and the unities had been populated in course of adding other elements, i.e. during calculation of hash functions for other keys (Bloom filter false positives). The reason for this is the property of HFs themselves - manifestation of collisions when two different arguments lead to the same hashing result.

Thus, the Bloom filter makes it possible to reduce the overall number of search queries to the data storage, and the main indicator of BF performance is probability of false positives (P). The smaller this indicator is, the less "needless" queries will be made to the physical media.

The probability of initially empty Bloom vector's i-th bit remaining equal to zero after adding N elements to the storage is represented by:

$$P_i = (1 - \frac{1}{M})^{KN} = e^{-\frac{KN}{M}}, \quad (1)$$

for a sufficiently large M, due to second distinctive limit.

The probability of all K-bits calculated by hash functions being equal to unity during searching the data storage for an element that is not equal to either of really added elements is:

$$P = (1 - e^{-\frac{KN}{M}})^K \quad (2)$$

From formula (2), we can conclude that with N increasing, the probability of false positives increases, while with M increasing, the number of errors of the first kind can be reduced.

For fixed N and M, the probability of false-positive result can be minimized by selecting K as follows:

$$K = \frac{M}{N} \ln 2 \quad (3)$$

In choosing the size of the vector, it is important to take into account the estimated number of elements to be stored and to initially set the probability of false triggering, then the length of Bloom vector can be calculated as follows:

$$M = -\frac{N \ln P}{(\ln 2)^2} \quad (4)$$

The Bloom filter can be easily set up using the above ratios. It makes it possible to reduce the load on the data access subsystem; however, it has one essential disadvantage. Elements cannot be removed from the classical Bloom bit vector. This means that it can be only used in systems where data is read-only; otherwise, the number of false positives will inevitably increase.

This problem can be solved by using a modification of Bloom filter with counters. In this approach, the Bloom vector is an array of L-bit counters that store values from 0 to 2L-1. When new data is added to the storage, the hash function calculates the value of the key and increments the relevant counter. In case the counter has reached the maximum value, its state does not change. In case of deletion, the counter is decremented by unity, or does not change its state if its value is zero or maximum. Due to such changes, the problem of deleting data can be solved.

Thus, implementation of a modified Bloom filter for supporting systems that store large amounts of information is an urgent task.

Modified Bloom filter operation principles

In design of a modified Bloom filter it is important to determine in advance the number of bits of its counter (L) and the size of the vector (M). With a fixed M, with increasing the bit number of the counter, the "capacity" limit is increased, i.e., the maximum value of the counter, starting from which the filter transfers to the classical operation mode (only for adding data). Increasing L by l leads to increasing vector size by M, resulting in a considerable increase in memory consumption.

One of the indications of modified Bloom filter correct operation is the fact that counters values differ insignificantly. This can be achieved by applying a qualitative hash function that provides a uniform distribution of its values for the entire range from 0 to M.

The qualitative hash function satisfies the suggestion of a simple uniform hashing, i.e., equiprobability of each key being placed into any of m cells independently of hashing other keys.

The universal hash function from book [4] was taken as the basis for the hash function that works with the Bloom filter.

Universal hashing is called the hashing method where instead of a fixed hash function, a random choice of a hash function occurs each time.

Building a set of the universal class of hash functions follows from the theory of numbers.

First, a prime number p that is large enough for all possible keys to be in the range $[0, p-1]$ is chosen.

Suppose $Z_p = \{0,1,..., p-1\}$, and $Z_p^* = \{1,2,..., p-1\}$. From the assumption that the key space is greater than the number of cells in the vector, it follows that $p > m$.

Now let us define a hash function h_{ab} for any $b \in Z_p$ and $a \in Z_p^*$, using linear transformation by absolute value p and then by absolute value m:

$$h_{ab}(k) = ((ak + b) \bmod p) \bmod m \quad (5)$$

The family of all such functions forms the set:

$$H_{pm} = \{h_{ab} : a \in Z_p^* \parallel b \in Z_p\} \quad (6)$$

Each hash function h_{ab} shows Z_p in Z_m. This class of hash functions has the feature that size m of the output range is arbitrary and is not necessarily a prime number. Since number a can be chosen by the $p-1$ method, and number b by p methods, the set H_{pm} totally contains $p(p-1)$ hash functions.

Within the framework of this experiment, it was decided to ignore storage type. Storage items in the Bloom filter were whole numbers generated with the Math.random() random function [5].

In the initial phase, an array was generated from N random elements. Elements of this array (keys) were added to the Bloom vector and the data structure implemented by the HashSet class. If overflow occurs in one of counters in the Bloom vector, the latter is converted into a bit one, and delete operations are prohibited. In the next phase, N queries to Bloom filter are executed, that had also been generated using the Math.random() function.

In accordance with the Bloom filter organization principles, the following situations may occur during its execution:

POSITIVE - The Bloom filter signaled the presence of a key, the key has been found in HashSet structure

FALSEPOSITIVE - The Bloom filter signaled about the presence of a key, the key has not been found in HashSet structure

NEGATIVE - The Bloom filter signaled about the absence of a key, there is no need to check HashSet

The testing program has counters for all above results. Experimental value of false positives probability can be calculated using the following formula:

$$RESULT = \frac{FALSEPOSITIVE}{POSITIVE + NEGATIVE + FALSEPOSITIVE} \quad (7)$$

Implementation of the modified Bloom filter using the NVIDIA CUDA technology

Tests were performed on a computing cluster of 12 computers having the following configuration: Intel Core i7 - 2600, 4Gb DDR3, NVIDIA GeForce GTX 260. The above video card supports CUDA Compute capability 1.3.

In order to speed up the Bloom filter, hash operations were executed on GPU. Coefficients from (5) were loaded into cached constant memory. Data for hashing were loaded into shared memory. After that, the necessary operations were performed to generate hash values, and results were loaded into global memory. Calculations were performed until all results were obtained. Ready values were passed in host memory, where they were interpreted as indices in Bloom vector.

Since Bloom filter itself, as well as the testing shell were written in Java, and jCuda library [6] was used to link the main program and the CUDA coprocessor, library which is a set of functions for compilation, transmission of parameters, configuring and calling the cuda-kernel.

Studying efficiency of the modified Bloom filter

Execution time of the sequential part of the program was measured on CPU by using the System.nanoTime() function that returns the current time value of a working Java virtual machine with high resolution capability, in nanoseconds. Subtracting the value before calculation from the value returned by this function after calculation, we obtain the execution time.

The execution time of the testing program was determined with the use of the graphical processor using API of CUDA events [7]. In CUDA, an event is a GPU timestamp recorded by a user at a certain moment.

Below are diagrams of some results obtained during the testing. From formula 2 it is obvious that with increasing number of added elements with predetermined number of hash functions and with fixed vector length, the probability of false positives will increase.

The diagram shown in Figure 5 fully corresponds to this formula (parameters: M=100, K=4, N=[1..1000].)

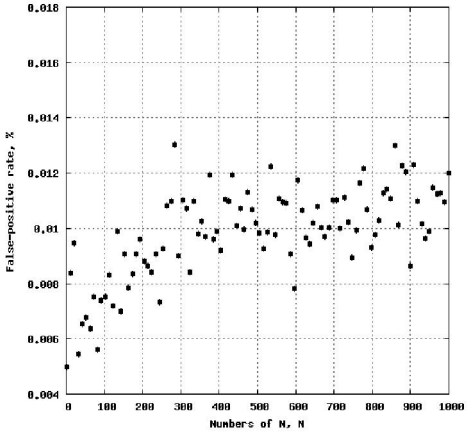

Fig. 5. Probability of false positives on the number of added data dependency diagram

False positives on vector length dependency diagram (Figure 6) show that for a fixed number of added elements, and for a fixed number of hash functions (on the diagram: N=1000, K=4, M=[1..100]), increasing the length of the Bloom vector reduces the number of false positives.

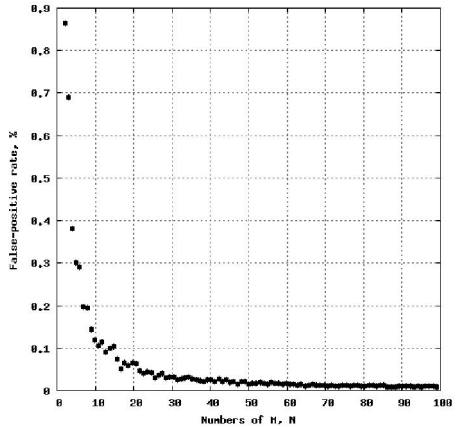

Fig. 6. False positives on Bloom vector length dependency diagram

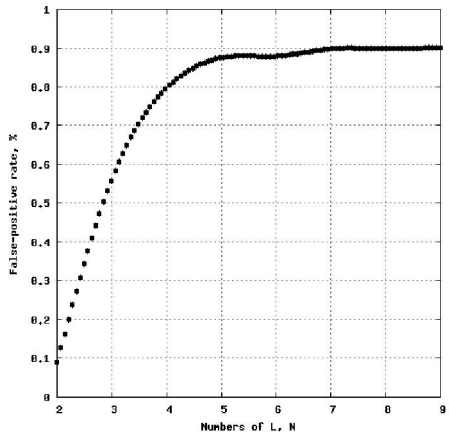

Fig. 7. Probability of false positives on counter's number of bits dependency diagram

From the diagram in Figure 7 it is evident that as early as at L=3, the percentage of false positives increases from 0.1 to 0.6, thus it follows that the bit number of the counters should be chosen precisely.

Use of the GPU co-processor makes it possible to achieve significant gain in performance. Figure 8 shows that parallel implementation (black line) is 15 times more productive than the serial one (red line). The achieved high efficiency is in general consistent with the results obtained by other researchers, for example, in [8, 9], but it should be noted that these works studied the Bloom bit vector. Fluctuations of the CPU diagram are associated with work of the Java "garbage collector" [10].

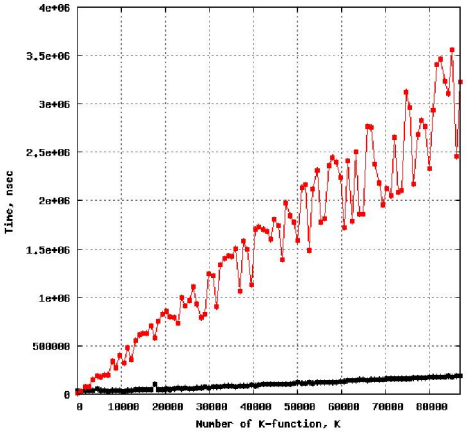

Fig. 8. Execution time on the hashed data volume dependency diagram

However, if the volume of hashed data is small, successive implementation shows better results due to absence of the extra need for copying data into the memory of CUDA device (Figure 9).

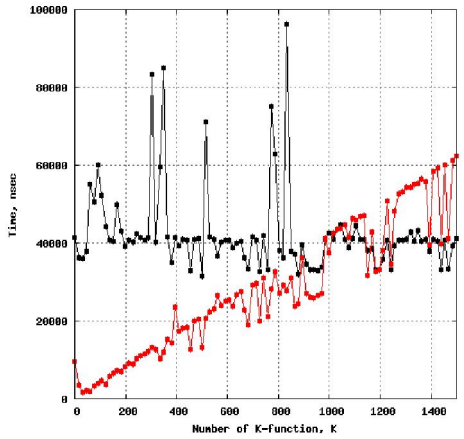

Fig. 9 Performance with small volumes of hashed data

Conclusion

Principles of Bloom filter operation have been studied, and its modification has been proposed. It makes it possible not only to add data, but to delete data as well. Main characteristics of the Bloom filter and their influence on quality of its operation have been described. Basic aspects of implementation of successive CPU and parallel GPU versions of the filter have been studied. Experimental diagrams have been analyzed, and growth of efficiency of a hybrid BF was detected in case of increasing the number of queries and data storage size, if compared to the CPU version.

The modified Bloom filter focused on hybrid computing systems can, according to the authors, be effectively used in modern supercomputers as a tool for cutting off many useless search queries in deployment of a variety of systems for storage and processing large volumes of data.

Corresponding Author:
Dr.Vavrenyuk A.B.
National Research Nuclear University MEPhI (Moscow Engineering Physics Institute)
e-mail:abvavrenyuk@mephi.ru

References

1. Worldwide LHC Computing Grid (official website). DateViews:02.04.2014 http://wlcg.web.cern.ch.
2. Glover, A., 2010. Java development 2.0: NoSQL. IBM Corporation, pp: 14.
3. Bloom,B., 1970.Space/time trade-offs in hash coding with allowable errors. Communications of ACM, 13(7): 422-426.
4. Cormen,Th.H. et al, 2009. Introduction to Algorithms, third edition. The MIT Press, pp: 1312.
5. Schildt, H. 2011. Java: the Complete Reference, 8thEdition. McGraw-Hill Osborne Media, pp.1152.
6. Java bindings for CUDA – Websitej Cuda. Date Views: 02.04.2014 http://jcuda.org.
7. NVIDIA CUDA C Programming Guide 5.5 (official website). Date Views: 02.04.2014 https://docs.nvidia.com/cuda.
8. Ma, L., R.D. Chamberlain, J.D. Buhler and M.A. Franklin, 2011. Bloom Filter Performance on Graphics Engines. In the Proceedings of the 2011 International Conference on Parallel Processing, pp: 522-531.
9. Rao, V., 2010. Implementation of the Bloom Filter on GPU using CUDA. University of Minnesota, pp: 6.
10. Hunt, Ch., 2011.Java Performance. Addison-Wesley Professional, pp: 720.

12/06/2014