Creating a Concurrent Overflowing Bloom Filter

Alex Berliner, Brian Estes and Ebin Scaria, University of Central Florida

Abstract—Bloom filters are an efficient probabilistic data structure used to verify membership of an element inside of a set. There is diminishing marginal value for inserting each additional element into a Bloom filter, and so steps must be taken to maintain scalability. One such option is to create a secondary hash set for a particular hash set in a Bloom filter that has become full, known as an overflow area. At this time, there are no implementations of a Bloom filter that implement this overflow system while maintaining concurrency.

In this paper, we demonstrate the creation of a concurrent overflow system for Bloom filters. We use the base Bloom filter presented by [1] and replace their method of dynamically resizing the Bloom filters with our overflow table implementation, as outlined in one of their suggested areas for future exploration. We then compare the results of our Bloom filter with those from the previously mentioned implementation as well as a standard Bloom filter.

Index Terms—Bloom Filters, Concurrency, Overflow Table

I. INTRODUCTION

Bloom filters are an efficient probabilistic data structure used to verify membership of an element inside of a set. When performing a contains query on a Bloom filter, the possible responses are either “this element may have been added” or “this element definitely has not been added”. A Bloom filter consists of many hash sets of the boolean type, each using a different hash function. In lieu of storing the elements added themselves, each set will hash the element being added to obtain a unique index to be set to true. [2] When multiple elements are added to a Bloom filter, some elements may hash into indices already set to true in a particular hash set. If Bloom filters were used to determine whether an element definitely had been added to a set, this system could result in false positives to a detrimental effect. However, because Bloom filters are designed only to be able to answer whether an element has not been previously added, this is considered to be a non-issue; the only guarantee that a Bloom filter makes is that it will maintain a 100% false negative rate.

Although a Bloom filter is designed to allow false positives, the false positive rate is inversely proportional to how useful the Bloom filter is; a Bloom filter with a high false positive rate will not be able to declare “this element has not been added to the filter” very often.

The primary benefit of using Bloom filters is to maintain high space efficiency. To achieve a false positive rate of only 1%, a Bloom filter must allocate less than 10 bits per element[2]. In some situations, this is much preferable to storing the element itself to achieve a perfect false positive rate of 0%, because the element can be of arbitrary size.

A. Problem

When enough elements are added to a fixed size Bloom filter that the false positive rate becomes unacceptably high, the Bloom filter is considered to be “at capacity”. The culprit here is that one or more of the hash sets that exist inside of the Bloom filter has already had many of the booleans that it contains set to true, so much so that adding further elements to these hash sets results in little or no new data being impressed upon the filter. This is a recipe for false positives as it is ineffectively attempting to store more data in the filter when it is not able. An overflow Bloom filter can be used to address this capacity concern by providing a secondary hash set to store elements in when the first is full, for each hash set in the filter. Although implementations of this type of Bloom filter exist, the group has not been able to find any that are designed to work in parallel, with either a lock or wait-free solution.

B. State-of-the-Art

Current research regarding Bloom filters are mostly topics that implement Bloom filters in creative ways or projects that aim to tailor and optimize the data structure for being used with specific applications[3]. We also looked at several hashing methods for reference before implementing this paper.[4], [5], [6]

One such optimization that has been researched recently is [7], where Bloom filters are being used to “store[e] topological information in large-scale mobile ad hoc networks.” These researchers have made a Bloom filter that is used to route large ad hoc networks efficiently by storing the network’s topological information with Bloom filters.

In terms of specifically tailored Bloom filters, networking is an area that gets a high amount of attention. The research done outlined in [8] for example is for a Bloom filter variant called a “distance-aware Bloom filter.” This filter works by using bitmasks to decay the information that exists in the Bloom filter based on its proximity to other Bloom filter nodes in a network.

A final example of a Bloom filter tailored to a specific purpose is the design of [9]. It uses a multiple Bloom filter (a layered Bloom filter) to attain a high compression rate for navigation data that is used for routing a wheelchair with a wireless camera.

C. Related Work

The current method for resizing a “full” Bloom filter is described in [1]. When this method is used on a non-blocked Bloom filter, the resulting data structure is called a Scalable Bloom filter. When the same method is used on a blocked Bloom filter, the resulting structure is called a
D. Contributions to the Field

We present one multifaceted contribution to the area of Bloom filters in this paper. As mentioned previously, the overflow Bloom filter mentioned in [2] could be used as another candidate for parallelization. At the time of writing, we could not find anywhere any implementation of an overflow Bloom filter at all. We are presenting an implementation of such a Bloom filter in this paper. In addition, the version of the overflow Bloom filter that we present is also designed to be used as wait-free data structure.[10], [11] This means that multiple threads may use this structure concurrently without waiting for another thread to complete its operation, and that each thread is guaranteed eventually to make progress.[12], [13]

In the event that a single block within a blocked Bloom filter becomes full, we have implemented a copming mechanism called an overflow table. This is rather than making the hash sets inside the Bloom filter scalable.

This paper aims to implement a wait-free version of the previously described overflow Bloom filter. Consider the following situation: for two threads A and B executing in parallel, A begins inserting an element into a specific block in the Bloom filter. It begins to check the current capacity of the block. Simultaneously, B finishes adding an element to the same block, which fills the last remaining space in that block. We want to ensure that A still receives an accurate capacity reading and places its element in the overflow buffer.

E. Our Algorithms & Approach, and Their Application

1) Deciding the Number of Hash Tables: Throughout a Bloom filter’s lifespan, the number of hashsets that it uses, or “layers,” is decided at the beginning of its execution. It is impossible to add another layer to a Bloom filter during its execution without significant overhead, as due to each layer using its own unique hashing function, the Bloom filter would have to hash each element inserted into the filter since the very beginning of its lifespan. This is an unreasonable scenario for most use cases for many reasons. For one, the program would have to keep track of the ordering of the insertion of the elements, which may generate an unreasonable amount of overhead for the program as a whole. Additionally, the act of reconstructing the Bloom filter layer itself will take time and may freeze the addition of more elements to the filter; a major problem when attempting to make a parallel data structure. These factors lead us to believe that it is more efficient to decide the right number of layers to create in the first place, and to never increase that number during execution. This comes at the cost of a dimension of scalability, as it means that the acceptable rate of false positives cannot change throughout the program’s lifecycle.

The algorithm that we used to decide how many layers to include in our Bloom filter is derived in [14] and is as follows, where:

\[ k = \log_2(P^{-1}) \]

In the case of this program, we have decided to use 1% as our acceptable false positive rate.

2) Capacity Formula: A Bloom filter’s main weakness is that it is prone to returning false positives when queried for a particular object. The more objects that are inserted into the Bloom filter, the higher the likelihood of encountering a false positive. There is a formula designed to calculate the number of elements that can be inserted into a particular Bloom filter before the likelihood of false positives becomes unreasonable. We will be utilizing it in our overflow Bloom filter to determine if a block has reached a capacity which requires an overflow table.

The method we use to determine whether a Bloom filter has reached capacity is presented and proven in [14]. The equation they use to determine the max number of elements that should be inserted into a Bloom filter is as follows, where:

- \( n \) is the number of elements inserted;
- \( M \) is the size of the filter in bits;
- \( p \) is the probability of a given bit being set in a hash set; and
- \( k \) is the number of hashsets to use.

\[ M = \left\lceil \frac{n}{n \ln(P)} \right\rceil \]

3) Generating Many Hash Functions: Originally, we implemented our hash tables such that each table based its size on a different prime number. This was an easy way to designate each hashset with a semi-random hashing function without having to generate an arbitrary number of hash functions as every different hash value will be mapped to a different index when modulus by the table size is performed. Although this method works for small test scenarios, it runs into problems when one is trying to tune the Bloom filter to different data sizes. It requires that the hash set sizes be set statically, or to use an algorithm to find many prime numbers to use to find table sizes for an arbitrary number of hash functions. This method we deemed to be too difficult to manage. Additionally, this technique results in certain Bloom Filters storing more information than others, which can lead to unnecessary complications. As a result, multiple complex hash functions would have to be implemented in overlapping layers in order to achieve the same level of randomness as using differently sized hash tables.

The authors of [15] have devised a way to achieve the same level of randomness as that of many hash functions using only two hash functions. As proven in [15] for two arbitrary hash functions, \( H_1 \) and, \( H_2 \), and a variable number \( i \), an arbitrary number of randomly distributed hash functions can be generated using the equation:

\[ H_i = H_1 + i \cdot H_2 \]

, where each value of \( i \) is a different hash function. In our overflow Bloom filter, this equation is used to generate the hash functions for the arbitrary number of hash sets that may potentially be used over the lifecycle of the Bloom filter.
4) Murmur Hash: The base hash functions that we used to generate our many hash functions for our hash sets is from a hashing algorithm known as Murmur hash. This type of hash function was designed by Austin Appleby and is detailed in [16].

F. Overflow Table Overview

There were two main goals in this project. The first goal was to implement the idea of the overflow Bloom filter whose method, as far as we can tell, was first mentioned in [1]. The second goal after implementing the overflow table was to modify the Bloom filter data structure so that it operates under some kind of concurrent condition;[17] be it lock-freedom or wait-freedom.[6], [18] This paper decided to focus on creating a lock-free algorithm due to perceived problems that could occur with a wait-free implementation.[19], [20], [21]

1) Overflow Table Addition Operation: The way that an element is sequentially added to an overflow Bloom filter is as follows. When the Bloom filter as a whole decides that an addition operation is to be performed, it will then issue the addition operation to each layer contained in the Bloom filter. Each layer will then hash the object that it has received to determine which index to insert the element into. It will then check if the element has been already set to true. If it has, there is no more work to be done and the operation returns. If the element has not been added, then the layer will first check itself to determine if it is already at max capacity. It does this by updating a counter owned by that layer keeping track of the number of flipped bits. It will then set the currently used set to full if the capacity is over half of the max capacity. If the filter is at capacity, instead of attempting to add it to the filter, it will check if that layer’s overflow filter has been created with a null check, as each hashset uses a linked list structure to store the link to the next overflow filter. The addition operation will restart at this point in the overflow filter.

2) Overflow Table Addition Parallelization: These are the modifications to the overflow Bloom filter to make the element addition operation parallelized. The addition operation was modified so that instead of using a counter to store the number of flipped bits, an O(k) search is used instead. Although this is an order magnitude slower than the original method, however it is much preferable to using shared counters in parallel, which represent a major bottleneck. Additionally, when the addition operation was converted to work in parallel, a global announcement table was implemented. When an addition operation notes that the first hashset is full but the overflow table hasn’t been created, the thread will attempt to check out the corresponding hashset in the announcement table. If it is successful, it will do another check to make sure that the overflow table was not created in the time between the null check and the announcement table being checked out. If it it successful, it will add the overflow table and add the pending element to the new hashset. If it does not succeed, the thread will wait until the hashset is removed from the announcement table, indicating that the overflow table has been created, and then add its element.

3) Overflow Table Contains Operation: The way that an element is queried in an overflow Bloom filter is as follows. The thread first checks if the index of a layer is true. If it is, then it continues to check each layer. If all the layers were set to true, then the operation returns that the element may have been added to the Bloom filter. If any of the layers do not contain the element, then a null check is performed to determine if an overflow filter exists. If it does not, then it’s confirmed that the element does not exist, and the Bloom filter returns accordingly. If the overflow filter does exist, then the primary filter will query it for the same element and it will return the result of that contains operation. This recursive method allows any number of overflowing hash sets to be made although many are not usually created over the course of a normal execution.

4) Overflow Table Contains Parallelization: There were not any changes to the code that had to be made to support parallelization. However, it is important to point out the linearization point of the operation as the lack of a definition of one for this operation can lead to perceived inconsistencies with the operation. The linearization point for contains is defined as the time when the boolean array is checked.

G. Ways that we combat concurrency problems

These are the main problems that we have had to overcome during the making of the overflowing concurrent Bloom filter.

1) Dynamic Creation of Additional Filters: Since each layer in the Bloom filter stores a link to another hashset that can potentially be set and accessed by multiple threads, precautions must be taken so that multiple threads do not create the same hashset simultaneously. For threads A & B and a hashset H0, this may arise when A sees that H0 has reached its maximum capacity and begins to create an overflow hashset called H1. Before the completion of the creation of H1 by thread A, thread B adds an element to H0 and also notices that H1 has not been created. At this point, thread A finishes creating H1 and adds it element to it. Afterwards, thread B finishes creating its version of H1, overwriting the H1 that thread A created. In this scenario, the information that thread A has committed to H1 has been permanently lost and represents a scenario where false negatives may be reported; a critical failure of the data structure’s integrity.

To remedy this, a single thread must take the burden of creating the overflow filter so that multiple threads do not try to create the same resource simultaneously in the form of an announcement table. This announcement table is an Atomic-Boolean array made from Java’s concurrency package. The parameters and usage for this array are as follows: A single announcement table exists for the entire scope of the Bloom filter. The announcement table contains an AtomicBoolean for each layer in the Bloom filter that dictates whether a thread is currently in the process of creating an overflow table for that layer. When a thread performs an addition to a hashset that is marked as full, it will attempt to perform a compareAndSet operation on that layer’s corresponding AtomicBoolean, marking it from false to true. If the set is successful, it means that the thread is clear to begin creating the overflow table. If
another thread tries to add elements to that hash table during the creation process, its compareAndSwap will fail, causing that thread to fall into a waiting state for the overflow table to be added, at which point the addition of that thread’s element will be retried. The addition of overflow tables is expected to happen fairly infrequently and is resolved quickly, so it is not considered to be a major bottleneck.

An example of this methodology for the concurrent operations takes place as follows: Consider threads A & B, hashset \( H_0 \), an uncreated overflow hashset linked to \( H_0 \) called \( H_1 \), and the AtomicBoolean linked to the hashset called L. Thread A attempts to add an element to \( H_0 \) and realizes that \( H_0 \) is full. Thread A checks to see if \( H_1 \) has been created, to which it sees that it has not. Thread A then does a compareAndSet operation to check out L. The check succeeds, and A then does another check on \( H_1 \)’s existence to make sure that it was not created by another thread entirely between the null check and L’s checkout. It succeeds, and so A is able to begin creating \( H_1 \). At this time, thread B then attempts to add an element to \( H_0 \). It sees that \( H_0 \) is full, and that \( H_1 \) has not been created. It then attempts to check out L, but it is unsuccessful, so it moves to a state where it waits until L has been set to false and \( H_1 \) is no longer null, which would indicate that \( H_1 \)’s creation has been finished. Thread A then finishes its creation of \( H_1 \) and unlocks L and then adds its element. B then notices that L is unlocked and \( H_1 \) is no longer null, at which point it is able to add its data as well.

2) Capacity Checking: While a thread is creating an additional Bloom filter for a hashset that has reached capacity, the operation of other threads must be halted to guarantee that more elements are not added to that hashset. We originally considered this the only implementation that degraded the algorithm to being only lock-free, as we thought other methods could be used to maintain the data structure’s wait-free nature, however we now believe that there is no way to implement this data structure in a wait-free way. Originally, we believed that we could make this structure wait-free by creating the overflow filters in advance so that whenever an overflow filter was needed it would already be created. This was implemented by setting a lower threshold for when a thread should begin creating a new overflow table than the threshold for what a thread considers to be at capacity. This method can be proven to be only lock-free using a direct proof. For threads A & B inserting into a hashset \( H_0 \) with an uninitialized overflow table \( H_1 \), A discovers that \( H_0 \) is full enough that \( H_1 \) must be created, to which it starts the creation process. Since \( H_0 \) is wait-free, thread B continues to add elements to \( H_0 \) until it is at capacity. At this point, since A is already in charge of creating the overflow table, B must wait to add its elements to the overflow table, destroying any wait-free guarantee.

3) Record Keeping: The capacity of the individual hash sets must be kept track of at all times to ensure that the overflow filter is created while maintaining the false positive guarantee. Originally, we thought that if each thread were to calculate the current fullness each time an element is added, it may cause an unneeded slowdown as the check turns an \( O(1) \) operation into an \( O(k) \) operation, where \( k \) is the number of elements in a single layer, as well as being inaccurate as other threads may be changing elements in the array as others are doing calculations. This may not matter depending on how the linearization point for the operation is defined but it is slow none the less. An alternative method is to keep a shared counter keeping track of the number of elements that have been changed in each hashset, however this will lead to the usage of highly contested resources, which is also undesirable.

The final method that the group decided on when developing the project was to perform the \( O(k) \) search. This method removes a significant bottleneck in the code an allows for parallel activity to continue even if the operation is lengthy. After work on the project had been completed, the group found an algorithm by Brian Kernighan that allows for the counting of the number of true bits in the array in \( O(\log(n)) \) time. Although we were not able to apply this optimization when doing the project, it represents an opportunity for future optimization.

H. Performed Experimental Results

The most important metric concerning the overflow table’s functionality is its ability to avoid false positives when more elements are inserted than the filter is expecting. Figure 1 shows that the bloom filter can maintain an average false positive rate of under 5 percent until the number of elements inserted becomes 5 times the number of elements expected. Additionally, it only returns a false positive at most 35 percent of the time even when the filter contains 40 times the amount of items it is intended to hold. To gather the rate of false positives, we ran a series of operations on the filter and a grow-able hash-table. Then we kept track of how many times each data structure returned that an item was not contained within it. Since the Hash Set is deterministic we know for a fact an item was not added to it, while the Bloom Filter is ambiguous. By subtracting the number of “does not contains” received from the Bloom Filter from the ”does not contains” received from the Hash Table we get the number of false positives received from the Bloom Filter. In order to maintain the order of the operations we had to run this in a non-concurrent implementation of our Bloom Filter.

Figure 2 shows the time required to insert a certain number of elements when the overflow table is enabled. This data was
recorded when two threads were running concurrently on the program.

Figure 2. Time Data for the Filter with its Overflow Table enabled.

Figure 3 shows the time required to insert a certain number of elements when the overflow table is not enabled. This data was recorded when two threads were running concurrently on the program.

When you compare figures 2 and 3, you can see that the time to insert a certain number of elements remains constant regardless of whether the overflow table is on or off. This implies that the time required to create the overflow tables when a particular hash set becomes full is negligible.

II. CONCLUSION

We consider the results of this project to be a success. We were able to successfully develop an implementation of the overflowing Bloom filter. Then, by analyzing the methodology that the data structure used, we were successful in developing a version that is able to work in parallel in a lock-free manner using core parallelization principles. Although this type of Bloom filter does not exhibit any increase in speed over a regular concurrent Bloom filter, this filter is able to deal with datasets of unknown size in a much better fashion than normal Bloom filters by being able to add additional storage space to the data structure on the fly. As mentioned in our results section, this implementation is able to store up to a factor of 5 times more data elements than a regular Bloom filter while still maintaining a false positive rate under 5 percent. When we began researching this topic, we theorized that it would be able to maintain an error rate of under 1 percent at such a size difference, but we consider that to be a somewhat unrealistic goal now. As can be seen from the results, trade-offs exist between using overflowing Bloom filters and Static Bloom Filters. Overflow Bloom Filters can hold more elements than intended, but only until a certain point. After the bloom filter contains 10 more items than their intended size, their false positive rate becomes unmanageable.

APPENDIX A

- Challenges
  - One of the major challenges when completing this project has been implementing the overflow table in an efficient way. A complete optimization pass for this data structure can be done which may warrant its own paper, but we have decided to focus on the parallelization only
- Set of Completed Tasks
  - Implemented simple Bloom filter (no blocking or resizing)
  - Implemented Parallelization techniques on the simple Bloom filter
  - Researched scalable, blocked, and dynamic-blocked Bloom Filter implementations
  - Researched and outlined concept of overflow-blocked Bloom filter
  - Implement the concurrent overflow-blocked Bloom filter
  - Parallelize overflow-blocked Bloom filter
  - Compare results to that of the dynamic hash table and regular Bloom filter

REFERENCES


