2015 International Conference on Cloud and Autonomic Computing

Idempotent Distributed Counters Using a Forgetful Bloom Filter

Rajath Subramanyam, Indranil Gupta, Luke M Leslie, Wenting Wang
Department of Computer Science
University of Illinois at Urbana-Champaign, Urbana, IL, USA

***Abstract*—Distributed key-value stores power the backend of high-performance web services and cloud computing applications. Key-value stores such as Cassandra rely heavily on *counters* to keep track of the occurrences of various kinds of events. However, today's implementations of counters do not provide exactly-once semantics. A typical scenario is that a client requests a counter increment, times out waiting for a response, and creates a duplicate request, thus resulting in a double increment on the server side. In this paper, we address this problem by presenting, analyzing, and evaluating a novel server-side data structure called the Forgetful Bloom Filter (FBF). Like a traditional Bloom filter, an FBF is a compact representation of a set of elements (e.g., client requests). However, an FBF is more powerful than a Bloom filter in two aspects: i) it can forget older elements (e.g., requests that are too old to apply), and ii) it is self-adapting under a varying workload. We also present an adaptive variant of FBF that adapts itself to meet a desired false positive rate – thus the error achieved in the counter can be bounded even as the workload changes. We present experimental results from a prototype implementation of FBFs and discuss the implications for a key-value store such as Cassandra. Our results show that the FBF is highly accurate in maintaining correct counter values.**

I. Introduction

Modern distributed applications are changing the landscape of commercial web-services. Distributed key-value/NoSQL stores like Cassandra [1], [2], RIAK [3], BigTable [4] and DynamoDB [5], [6] are becoming increasingly popular due to their higher availability, scalability, and performance compared to traditional relational databases.

In today's key-value/NoSQL systems, a *counter* [7], [8] is a commonly used type of data structure (usually a column in a database table). Counters support increment (e.g., +1) operations from multiple clients and are used for counting the number of tweets/likes/retweets, number of ad clicks, or system parameters like the number of times an object was accessed [9], [10].

Applications using key-value/NoSQL stores require a guarantee of *idempotence*, i.e., that operations such as counter increments be carried out exactly-once. However, it is well-known that exactly-once semantics are hard to achieve in asynchronous systems [11]. For instance, when a client sends an operation to the server, the server might fail after applying the operation but before sending back an ack. In this case, the client has no way to know whether the operation was applied or not [12]. In such a scenario, sending a duplicate request might lead to an over-count, whereas not sending one might lead to an under-count if there was indeed a server failure.

Today, there are three potential approaches to solving this problem of idempotence in key-value stores such as Cassandra:

1) Clients do not submit duplicate requests, resulting in at-most-once semantics.
2) The server maintains a list of all updates that have been applied to an object (e.g., the counter) so that duplicate operations can be checked and rejected.
3) The server utilizes a Bloom filter [13], [14], a probabilistic data structure of fixed size, to store updates. A Bloom filter is a compact representation of a set of elements, and allows fast checking of membership in the set. Each client operation is uniquely timestamped (e.g., by using the client id and a sequence number) and added to the Bloom filter when the operation is performed. Subsequent duplicate operations can be checked quickly against the Bloom filter and applied only if not present.

We discuss these approaches in some detail. Approach (1) is the default in Cassandra. This approach is cheap, but it has the disadvantage of losing requests when there is a failure of the server or message delivery and thus results in under-counts. Approach (2) is correct and has exactly-once semantics (eventually), however, it is very costly in the long run – the history of operations can grow unboundedly and checking for duplicates becomes prohibitively expensive.

Approach (3) takes advantage of the fact that adding an element into a Bloom filter is an idempotent operation; repeating addition of a given element gives the same result. Thus, servers can be sent duplicates of the same entry and the Bloom filter will reflect only one copy. At the server side, determining if an operation has been performed is reduced to a simple membership check. However, as the number of elements inserted into a Bloom filter increases, so does the probability of a *false positive* (i.e., membership checks may erroneously return true). False positives result in an under-count. Using larger Bloom filters will only delay the problem. Further, in Bloom filters it is impossible to: 1) delete entries from the Bloom filter, 2) grow the filter in size to scale with the number of elements, or 3) have entries timeout, from within the filter, without adding significantly more data (such as buckets [15]).

The problems associated with the above approaches motivate us to invent and introduce a novel data structure called the *Forgetful Bloom Filter (FBF)*. Like a traditional Bloom filter, an FBF supports an element addition operation that is idempotent. An FBF allows insertion and membership-

This work was supported in part by the following grants: NSF CNS 1409416, NSF CCF 0964471, NSF CNS 1319527 and AFOSR/AFRL FA8750-11-2-0084.

checking of items with the same asymptotic cost as a Bloom filter. Unlike a Bloom filter, an FBF takes advantage of temporal locality that exists in counter operations, i.e., storing old updates becomes less valuable over time as these operations become increasingly unlikely to be retried. In fact, clients often have write request timeout beyond which they give up on the write. Thus, an FBF automatically expires older items (the timeout period can be adjusted). An FBF does so by using multiple constituent Bloom filters to essentially maintain a *moving window* of recent operations.

Like the traditional Bloom filter, false positives can occur in the FBF, thus leading to an under-count for counters. However, unlike the Bloom filter, this false positive rate can be bounded in an FBF. We present adaptive techniques to grow and shrink the FBF, which allow it to always meet a user-specified upper threshold of false positive rate. Thus the error achieved in the counter can be bounded even as the workload changes.

In this paper we use counters as an exemplar application to demonstrate that the FBF approach provides probabilistic accuracy very close to approach (2) detailed earlier (exactly-once semantics) at roughly the same cost and with higher accuracy than approach (3) above (pure Bloom filters).

II. Background

In this section, we discuss the assumptions and provide a brief overview of Bloom filters.

A. Assumptions

We make the following assumptions:

- Data (e.g., key-value pairs) is stored and replicated at one or more servers.
- Clients issue operations via a front-end server (called a *Coordinator* in Cassandra) which then communicates with the replicas.
- All client operations are globally and uniquely identified, e.g., by using the tuple <client id, per-client sequence number>.
- Clients have a *write request timeout period* for each write operation. If a client does not get a response from the coordinator within this time period, the client aborts the write operation.
- There is a user-specified upper bound on the false positive rate of the data structure that we use to maintain idempotence. For instance, for counters, the user may specify an error percentage in the value, and the false positive rate can be derived directly from it.

B. Bloom Filters

A Bloom filter [13], [14] is a space-efficient data structure that can be used to capture elements of a set and check quickly for membership in that set. A Bloom filter is an array of m bits, each bit initially set to 0. When an element is inserted it is hashed using k (fixed) hash functions. Each hash output maps to a bit in the Bloom filter, which is then set. To check for membership, the same k hash functions are used to check if all the mapped bits are already 1. While Bloom filters always return the right answer for an element already in the set (no false negatives), false positives may occur as a non-present element may check to true because of the way bits were set. However, the rate of false positives can be lowered arbitrarily close to 0 by changing parameters such as k, m, etc. [13], [14].

Bloom filters are not ideal to solve counter idempotence for two reasons: i) bits set by older operations cannot be deleted or expired automatically from the Bloom filter, and ii) Bloom filter parameters (m, k) need to be fixed at creation time. Thus, once the Bloom filter fills up, it is hard to "scale up" the Bloom filter, or "move" its elements to another Bloom filter with different parameters unless one maintains all updates. As a result, over time, as more and more bits become set to 1, the false positive rate rises above the user-specified false positive threshold for that application.

III. Forgetful Bloom Filter

We present a new data structure that builds on the Bloom filter but also overcomes many of its shortcomings listed in Sec. II-B. Our new data structure is called a *Forgetful Bloom Filter* (abbreviated henceforth as *FBF*). An FBF maintains a moving window over recent operations. Thus it allows older operations to automatically time out and be deleted from the FBF. We also show how to keep the false positive rate below the user-specified threshold, even as the workload changes, by adapting the parameters of the FBF.

A. Basic FBF Operations

In its simplest form, an FBF contains three Bloom filters: i) a future Bloom filter, ii) a present Bloom filter, and iii) a past Bloom filter. All these Bloom filters are equal in size and identical in their use of hash functions. Such an FBF with only three Bloom filters is called a *Basic FBF*.

When an element needs to be inserted (Algorithm 1), it is first checked for membership in the FBF (we will describe the membership check in the next paragraph). If it is not present, it is inserted only into the future and present Bloom filters, but not in the past Bloom filter.

A membership check can be performed by checking if the tested element is present in at least one of the three constituent Bloom filters – if so, the check returns true. If the element is absent in all the three constituent filters, it is considered to be not present in the FBF. Additionally, the structure of the FBF allows us to optimize the membership check further, thus lowering the false positive rate – we describe this optimization in Sec. III-D.

When applied to the counter idempotence problem, recall from Sec. II-A that we assume that all client operations are globally and uniquely identified, e.g., by using a <client id, per-client sequence number>. Thus, when a server receives a client operation (such as a counter increment), it is first checked for membership in the FBF. If it is not present, it is inserted into the FBF and the operation is reflected into the database table. If it is already present in the FBF, the operation is discarded; however, another acknowledgment is returned to the client.

Algorithm 1 Insert element el into the FBF

```
procedure INSERT($el$)
    if $!membershipCheck(el)$ then
        $FBF[future].set(el)$
        $FBF[present].set(el)$
    end if
end procedure
```

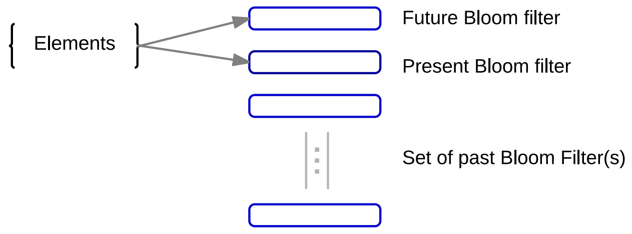


Fig. 1: An FBF is implemented as a list of Bloom filters. Elements are inserted into the future and present Bloom filters.

B. Generalized FBFs: the $N-$FBF

In general, our FBF may contain more than three constituent Bloom filters. Broadly, we say that an $N-$FBF contains one Future Bloom filter, one Present Bloom filter, and $N(\geq 1)$ Past Bloom filters. This is illustrated in Fig. 1. Our basic FBF becomes a special case – it is a 1-FBF ($N = 1$).

Insertion of a new element in an $N-$FBF affects only the future and present Bloom filters, but not any of the N past Bloom filters. The membership check remains unchanged (Sec. III-D).

C. FBF Refresh

In order to forget older elements, periodically (every t time units), an FBF undergoes a *refresh* operation. In a basic FBF, at a refresh point, the following operations are performed atomically:

1) The past Bloom filter is dropped;
2) The current present Bloom filter is turned into the new past Bloom filter;
3) The current future Bloom filter is turned into the new present Bloom filter;
4) A new, empty future Bloom filter is added to the FBF.

For an N-FBF, the refresh operation cascades intuitively through the past Bloom filters, from the newest to the oldest, as shown in Fig. 2. The oldest Bloom filter is dropped, the next oldest Bloom filter becomes the oldest, and so on. Finally, the old present Bloom filter becomes the newest past Bloom filter, the old future becomes the new present, and a new and empty Bloom filter is added as the new future Bloom filter.

Fig. 3 demonstrates the long-term behavior of a basic FBF due to its refresh operation. The refresh period is t time units. The time slices of data stored by each constituent Bloom filter is shown in the box representing the Bloom filter. A basic FBF stores client operations that have come in the past $(2t, 3t]$ time

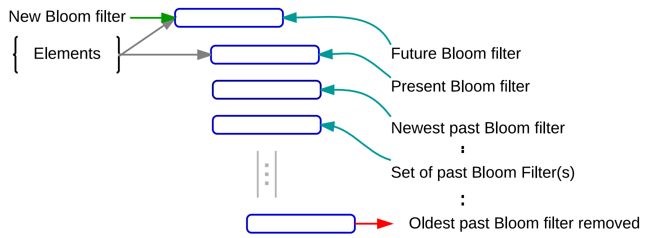


Fig. 2: A refresh operation in a $N-$FBF adds a new Future Bloom filter, removes the oldest Past Bloom filter), and moves the other Bloom filters up.

units. In general, an $N-$FBF stores client operations from the past $((N+1)\cdot t, (N+2)\cdot t]$ time units. We set this interval to be greater than the counter write request timeout period (which is typically configured in the key-value store clients based on suggested configuration parameters), beyond which the client will stop retrying the update operation. This ensures that the FBF captures all relevant operations, and that older forgotten operations will not be retried by clients.

Implementation-wise, the refresh operation of the FBF is carried out by a thread running in the background based on the refresh period set by the application. This operation is carried out seamlessly exposing only the standard set and get interface like a normal Bloom filter.

D. Optimized Membership Check

When checking an FBF for membership against an element in Sec. III-A, our approach was merely checking if any of the constituent Bloom filters contain the element. However, the false positive rate of the FBF can be further lowered by observing the following critical property of an FBF.

Property 1: In a basic FBF, the future and past Bloom filters do not overlap in time.

This property arises because the point of time at which a new (empty) future Bloom filter is created is at the same time instance when a past Bloom filter became one (it was a present Bloom filter before) – this is the atomic refresh point described in Sec. III-C. Since no further elements are inserted into a past Bloom filter, there is no overlap in time with the future Bloom filter. This is also pictorially evident in Fig. 3. More generally, we can show that:

Property 2: In an N-FBF, only consecutive (neighboring) Bloom filters overlap in time.

This leads us to the following optimization: we only need to check for membership simultaneously in pairs of overlapping Bloom filters. For instance, in a basic FBF we need not check to see if element is present both in the future Bloom filter and the past Bloom filter. This is because the element was first inserted into the FBF at some point of time – it could only have been inserted into either the current future filter or the current past filter, but not into both. On the other hand, if the element is in both future and present Bloom filters, or both present and past Bloom filters, or just the past Bloom filter then the element is genuinely present in the FBF.

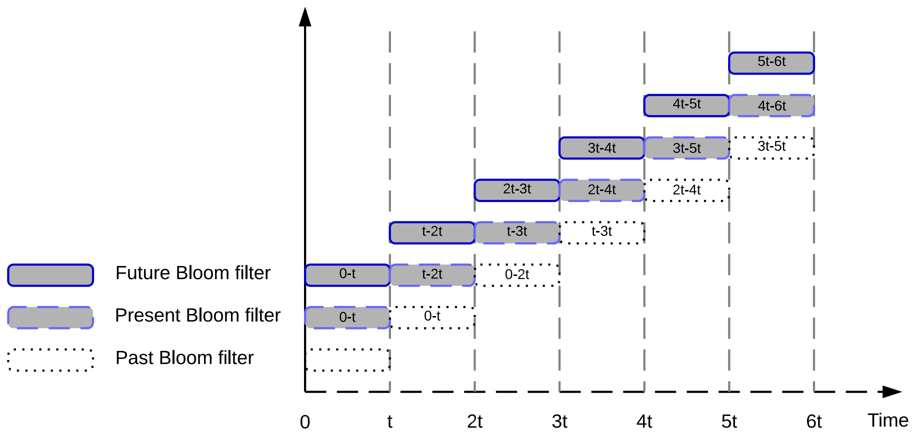


Fig. 3: Timeline of a basic FBF showing the refresh operation. The constituent filters with the darker shade represent the filters into which elements are inserted.

This leads us to an optimized algorithm for checking membership in an FBF. Algorithm 2 describes the pseudocode, and we explain below.

Consider an N-FBF with the future Bloom filter indexed as $future$, present Bloom filter indexed as $present$ and the past Bloom filters indexed from $pastNewest$ to $pastOldest$. Using Property 2, the algorithm proceeds as follows: the membership check proceeds by checking for membership in the most recent window of time until the oldest window of time is reached. If the membership check passes in any window of time (i.e., two consecutive Bloom filters return true), then there is no need to check further and the check can return true. Accordingly, the future Bloom filter is checked first. If it is not present there, the present as well as the pastNewest Bloom filter are checked together. Intuitively, the membership check proceeds with the next consecutive pairs of Bloom filters. Finally, the oldest past Bloom filter is checked. If all tests return false, then the membership check returns false (element is not present). The optimized version takes Property 2 into account by only checking Bloom filters that overlap in time.

Algorithm 2 Optimized Membership check for element el

```
procedure MEMBERSHIPCHECK($el$)
    if $FBF[future].get(el)$ then
        return $true$
    else if $FBF[present].get(el) \wedge FBF[pastNewest].get(el)$ then
        return $true$
    else if $pastOldest > pastNewest$ then
        for $i = pastNewest$ to $pastOldest - 1$ do
            if $FBF[i].get(el) \wedge FBF[i+1].get(el)$ then
                return $true$
            end if
        end for
    else if $FBF[pastOldest].get(el)$ then
        return $true$
    else
        return $false$
    end if
end procedure
```

E. Adaptation via Dynamic Resizing

As the workload changes over time, the rate of updates may go up (or down). This may lead to the constituent Bloom filters of the FBF filling up quicker (or slower) than the refresh rate can keep up. As a result, the false positive rate may spike.

In order to maintain the FBF's false positive rate below a user-specified threshold (e.g., based on the desired accuracy of counter operations), we show how to adapt the FBF parameters. Concretely, we adapt the following two parameters:

1) Number of constituent Bloom filters, and
2) Refresh rate.

Dynamic resizing is implemented by a thread running in the background. The thread calculates the false positive probability (see Sec. IV) and, as this value approaches the threshold set by the application, it adapts the number of constituent Bloom filters and the refresh rate. Again, this thread runs in the background seamlessly to the application exposing only the standard set and get interface like a normal Bloom filter.

The pseudocode is shown in Algorithm 3. The thread running every one second computes $curFPP$, i.e., the current false positive probability of the FBF (based on the analysis shown in Sec. IV). The $curFPP$ is then compared with the threshold false positive probability set by the application i.e., $targetFPP$. Then the approach uses multiplicative increase/additive decrease of the parameters. Broadly, when the false positive rate becomes too high and risks violating $targetFPP$ (e.g., due to an increase in volume of incoming requests), the dynamic resizing algorithm does two things:

1) It increases the number of Bloom filters in the $N-$ FBF from $(N+2)$ to $2 \cdot (N+2)$, by adding the extra Bloom

Algorithm 3 Dynamic resizing

```
procedure TRIGGERDYNAMICRESIZING( )
    if $curFPP() \geq 0.9 \cdot targetFPP()$ then
        //Multiplicative Increase of number of Bloom filters and Additive decrease of refresh period
        $scaleUpFilters()$
        $decreaseRefreshPeriod()$
    else if $curFPP() \leq 0.1 \cdot targetFPP()$ then
        //Additive Decrease of number of Bloom filters and Additive increase of refresh period
        $scaleDownFilters()$
        $increaseRefreshPeriod()$
    end if
end procedure
```

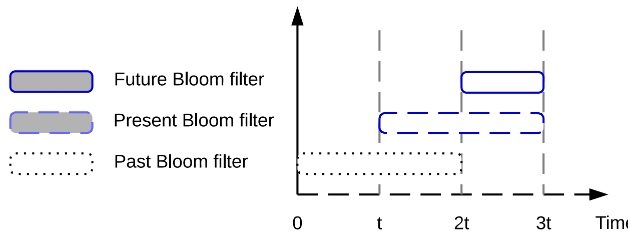


Fig. 4: The state of a basic FBF from 0-3t.

filters to the set of past Bloom filters.

2) It decreases the refresh period from t units to $(t-1)$ units.

This multiplicative increase/additive decrease action lowers the false positive rate by increasing the window size of past requests and filling up Bloom filters less. If it is still too high, further resizing operations may be initiated by the background thread.

Later, if the false positive rate subsides (e.g., due to a natural lowering of the update rate from clients), then the adaptive algorithm automatically adjusts these two parameters in the reverse fashion, i.e., the number of Bloom filters is decreased and the refresh period is increased. Both parameters are adjusted additively in order to be conservative. This ensures that the $N-$FBF stays as small and incurs as low overhead as possible, in order to support the current workload, at all times.

We adopt this approach inspired by TCP congestion control [16], so that we respond aggressively when the false positive probability is about to be violated, but let up gently when the situation starts returning to normalcy.

IV. Analysis: False Positive Probability of an FBF

We now derive an equation to estimate the false positive probability of an $N-$FBF consisting of N constituent past Bloom filters (and thus $N+2$ total Bloom filters). This equation is used by the dynamic resizing thread of Sec. III-E.

To make the analysis tractable, we assume: 1) hash functions are uniform, and 2) the Bloom filters inside an FBF behave independent of each other, as far as false positive rates are concerned. We later show through our experiments in Sec. VI that, despite our assumption of independence, our analysis holds in practice.

Consider a single Bloom filter B with m bits, k hash functions, and with l elements inserted. From, e.g., [17], the probability p of a false positive during a membership check on this Bloom filter is equal to the probability that all k bits corresponding to that element were set by previous insertions. This probability is thus equal to:

$$p(B) = \left(1 - \left(1 - \frac{1}{l}\right)^{mk}\right) \approx \left(1 - e^{-kl/m}\right)^{k} \quad (1)$$

To determine the probability of a false positive in an FBF with an arbitrary number of constituent filters, we first present the derivation of this quantity for the most basic $1-$FBF with three filters (future, present and a single past filter).

Consider a $1-$FBF with three Bloom filters, each with m bits and k hash functions as shown in Fig.4. In our $1-$FBF example, the membership check from Algorithm 2 first evaluates the future Bloom filter alone, then the present and past filter, and finally the past filter alone. Following the path for a membership check in our FBF, we therefore wish to determine the probability of some false positive in either: 1) the first Bloom filter (future), or 2) both the present and past filters, or 3) in the past filter by itself.

Let the past Bloom filters be indexed from most recent to oldest (i.e., $pastNewest = 1, \cdots, pastOldest = n$). Under the assumption of independence, the probability of a false positive in a $1-$FBF can be estimated as:

$$p(BasicFBF) = 1 - (1 - p(B_{future})) \cdot (1 - p(B_{present}) \cdot p(B_1)) \cdot (1 - p(B_1)) \quad (2)$$

From Equation 1,

$$p(B_i) = \left(1 - e^{-kl_i/m}\right)^{k}$$

where l_i is the number of elements inserted into Bloom filter B_i.

Generalizing Equation 2 to an $N-$FBF with $(N+2)$ total filters, we can therefore estimate the probability of a false positive as the probability of some false positive in either the first Bloom filter, both filters in some contiguous pair in the remaining filters, or in the last filter by itself. That is, for an $N-$FBF that holds updates over $(N+2)$ time intervals, we can determine the false positive probability within each interval (t_i, t_{i+1}) of the membership check as:

$$\phi_{i\to i+1}(N{-}FBF) = \begin{cases} p(B_{future}), & \text{if } i = n+1 \\ p(B_{present}) \cdot p(B_1), & \text{if } i = n \\ p(B_{n+1-i}) \cdot p(B_{n-i}), & \text{if } 1 \leq i \leq n-1 \\ p(B_n), & \text{if } i = 0 \end{cases}$$

The probability that some stage of the membership check will return a false positive is therefore equal to:

$$p(N-FBF) = 1 - \prod_{i=0}^{n+1} (1 - \phi_{i \to i+1}(N-FBF)) \quad (3)$$

Later (Sec. VI-B) we will compare this estimate with the observed false positive rate.

V. System Design: Integrating FBFs into Cassandra

In this section, we discuss how FBFs can be integrated into the partitioned counters of today's distributed key-value stores like Cassandra. Subsections $A - C$ below describe the background, and in D we show our integration.

A. Cassandra

While our techniques are applicable to any key-value store, for concreteness, we apply these to Apache Cassandra [1], [2]. Here we describe the background for Cassandra and its counters.

Cassandra is a decentralized, distributed key-value store where server nodes are arranged in a virtual ring using consistent hashing [18], and these servers store the corresponding key-value pairs. Keys are assigned to servers based on consistent hashing. Keys can be replicated at multiple server nodes for fault-tolerance, e.g., neighbor servers on the ring. Cassandra supports concurrency with multiple clients. Clients can send CRUD (Create, Read, Update, Delete) operations to any server node in the ring. This contacted server is called the coordinator for that operation. The coordinator node, upon receiving a client request, locates the replica node(s) responsible for that key via consistent hashing, forwards them the query, receives their response, and forwards it back to the client.

B. Counters

Counters [19] in Cassandra are a special-kind of data structure, typically stored as a special column inside a database table. Once a counter column is defined inside the table, clients can issue update operations (e.g., +1's) on any key in that table. The flow of a counter update operation is shown in Fig. 5 (it is slightly different from a normal update). When the coordinator receives a request from a client, it locates a suitable replica (of that key) as the leader replica and forwards the request to it. The leader replica forwards the counter update to other replicas. Each replica, after committing the operation to the commitlog [20] and the in-memory Memtable [21], sends back a corresponding ack to the leader replica which further sends back an ack to the coordinator, which in turn returns an ack to the client.

In Sec. V-C, we introduce the internals of partitioned counters design in Cassandra, their evolution, and their associated problems. In Sec. V-D, we show how FBFs can be integrated into distributed key-value stores like Cassandra and overcome the inherent problems to obtain idempotent, lock-free, correct, fast, distributed counters.

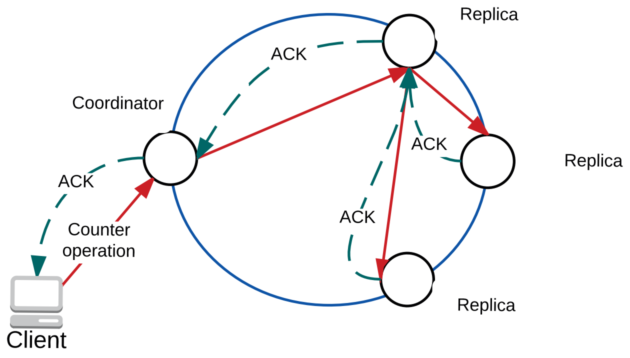


Fig. 5: Update operation on a counter with a replication factor (RF) of 3

C. Background: Partitioned Counters in Cassandra

Cassandra ($v0.8$ onward) supports *partitioned* or *distributed* counters [22]. Such counters are split and replicated across across multiple nodes (servers), thus ensuring fault-tolerance, availability, load-balancing, and scalability (i.e., if the replication factor or RF is > 1).

Each sub-counter (counter at a server) tracks all its received updates/deltas. When needed various anti-entropy techniques are used (e.g., read-repair [23], AES [24] and repair-on-write) to aggregate all updates for the counter across the cluster. The goal is to provide eventual consistency [25], i.e., the replicas eventually learn about all the updates in the system. This design of counters is not idempotent. Client-retries after a counter write request timeout or commitlog replays can lead to incorrect value of the counter. This is true for partitioned counters design used in all versions up to Cassandra $v2.0$.

Internally at each server, each sub-counter is split into fragments called 'shards'. A shard is a 3-tuple consisting of (node's counter id, shard's logical clock, shard's value) as shown in Fig. 6. The counter id is a value uniquely identifying the node that created the shard. The logical clock is a monotonically increasing number representing the number of operations committed on the shard. Finally, the shard's value is either the size of the increment or the total value, depending upon the shard type. A shard type could be either a local shard or a remote shard. Local shards track the updates/deltas received by this particular sub-counter. It is not possible to simply replicate this delta in other replica nodes. Instead, the local shards are summed and the total is sent to other replicas. Whenever a local shard is sent to another node, it is stored as a remote shard at that other node.

The final value of the counter at any replica node is the sum of the values of: 1) all the local shards in that node, and 2) that single remote shard in that node which has the highest logical clock.

We use an example to show the working of Cassandra's partitioned counters, as well as to show how a message loss can violate idempotence. This is depicted in Fig. 6. In this figure, consider a counter $c1$ with a replication factor of 2. The counter is replicated at node A and node B. In step 1, node A receives an increment of $+1$ to the counter $c1$. After committing the update to it's own commitlog and Memtable, the update is propagated to node B where it is appended to the commitlog and stored as a remote shard in B's Memtable. A client issuing a read of $c1$ at this point reads a value of 1 irrespective of whether the coordinator redirects the read request to node A or node B. In step 2, node B receives an increment of $+2$ to the counter $c1$. After committing locally, the update is propagated to node A where it is appended in the commitlog and updated in the Memtable as a remote shard. A client issuing a read of $c1$ at this point would read a value of 3 (i.e., sum of all local shards plus the that remote shard with the largest logical clock). At step 3, the Memtable is flushed to an on-disk SSTable [26] in both node A and B. In step 4, node A receives an increment of $+3$. After committing locally, the update is propagated to node B. At this stage, if the ack to the client were to be dropped by the network, the counter operation is replayed in step 4 when the client re-transmits its request – this immediately leads to an over-count. A client read of $c1$ will return 9, instead of the correct value of 6.

To deal with these issues, from Cassandra $v2.1$ onward, the concept of local shard and remote shard were eliminated by the developers instead, a (local) lock is acquired for each counter being updated, the local sub-counter value is read, and then the incremented value (not the increment itself as in Cassandra $v2.0$) is written into the commitlog and the Memtable and sent to the other node. The sum of all the local shards of a node need not be done. The value of the sub-count with the highest logical clock is chosen. This simplified design, however it has performance implications due to the lock-read-write-unlock. This new design of partitioned counters in $v2.1$ ensures that counter operations are idempotent in case of commitlog replays, but can still lead to incorrect values in cases of client retries after counter write request timeout.

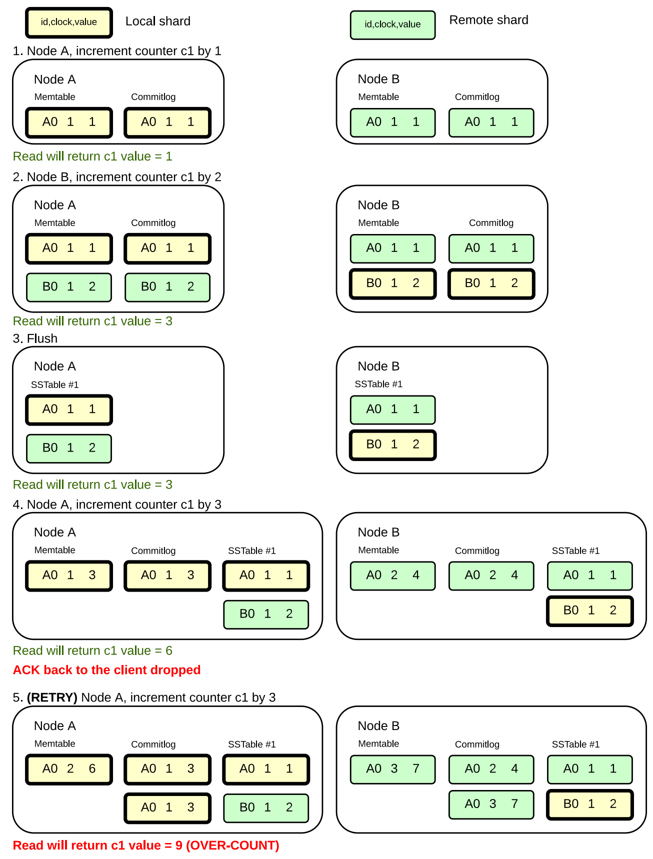
D. Idempotent Counters in Cassandra using FBF

In order to integrate FBFs with distributed key-value stores to obtain idempotent counters, we propose the following changes at the server-side and the client-side.

As mentioned earlier, each client operation needs to be identified by a globally unique identifier, such as <client id, per-client sequence number>. This involves minor changes to the client-side drivers.

On the server side, an FBF is associated with each counter column. In most key-value stores developed using object-oriented principles, this involves adding an object of the FBF implementation as a member of the counter column class.

The counter update (with an unique id) on each node containing the counter first goes through the FBF associated with that counter. A membership check is run in order to determine if that particular counter operation was previously applied or not. If the membership check passes, then it is dismissed as a retry; else it is appended to the commitlog and updated in the Memtable. All these steps are atomic. As long



id,clock,value
Local shard
id,clock,value
Remote shard
1. Node A, increment counter c1 by 1
Node A
Memtable
Commitlog
A0 1 1
Node B
Read will return c1 value = 1
2. Node B, increment counter c1 by 2
B0 1 2
Read will return c1 value = 3
3. Flush
SSTable #1
Read will return c1 value = 3
4. Node A, increment counter c1 by 3
A0 1 3
A0 2 4
Read will return c1 value = 6
ACK back to the client dropped
5. (RETRY) Node A, increment counter c1 by 3
A0 2 6
A0 3 7
Read will return c1 value = 9 (OVER-COUNT)

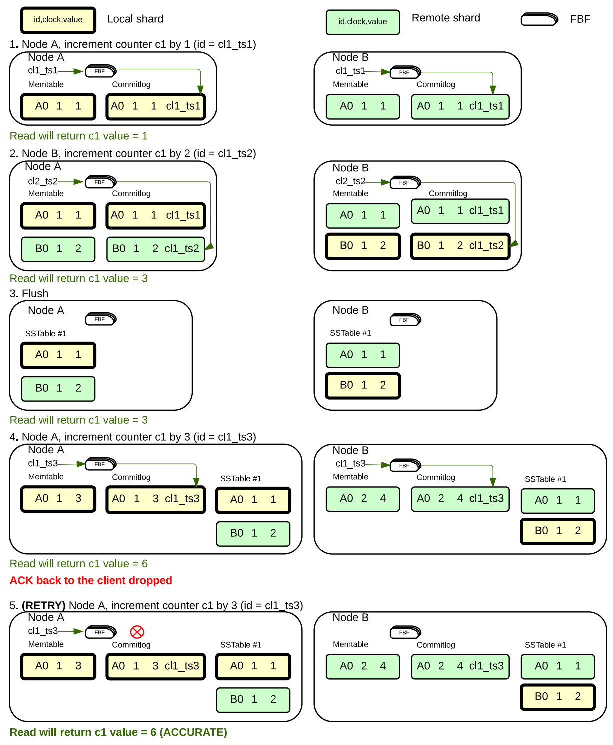
Fig. 6: An example depicting how Cassandra's partitioned counters work, and how a re-transmission of a counter update can lead to an over-count.

as $((N+1)\cdot t)$ (where N is the number of past Bloom filters in the FBF and t is the refresh period) is greater than the client counter write request timeout, retries will be dismissed. In the event of a false positive, a valid non-duplicate counter operation may be dismissed as a duplicate. However, this is a low probability occurrence. We have shown earlier (Sec. III-E) how FBF is capable of self-adapting to the workload in order to lower the false positives. We also show in Sec. VI that the number of false positives is very small.

Counters are replicated on multiple nodes to provide fault-tolerance, scalability and high-availability. The counter operations after being committed locally should be propagated to other nodes holding the replica of the counter. This ensures that the corresponding FBFs are synchronized and consistent across the cluster.

The shards stored in the commitlog also additionally contain the unique id associated with the counter operation in order to ensure that the updates are not duplicated during commitlog replays.

We chose the approach of associating FBFs with each counter column, rather than at the system level or the key granularity. A system-level FBF would have to handle too



id,clock,value
Local shard
id,clock,value
Remote shard
FBF
1. Node A, increment counter c1 by 1 (id = cl1_ts1)
Node A
cl1_ts1
Memtable
Commitlog
A0 1 1
A0 1 1 cl1_ts1
Node B
Read will return c1 value = 1
2. Node B, increment counter c1 by 2 (id = cl1_ts2)
cl2_ts2
B0 1 2
B0 1 2 cl1_ts2
Read will return c1 value = 3
3. Flush
SSTable #1
Read will return c1 value = 3
4. Node A, increment counter c1 by 3 (id = cl1_ts3)
cl1_ts3
A0 1 3
A0 1 3 cl1_ts3
A0 2 4
A0 2 4 cl1_ts3
Read will return c1 value = 6
ACK back to the client dropped
5. (RETRY) Node A, increment counter c1 by 3 (id = cl1_ts3)
Read will return c1 value = 6 (ACCURATE)

Fig. 7: An example depicting how an FBF integrated into Cassandra's partitioned counters can avoid over-count when there is a re-transmission of a counter update.

many counter operations (across multiple tables). A key-level FBF would create too many FBFs. The column-level (or rather table-level) FBF gives the best of both worlds. This approach also entails minimal changes to the underlying design of key-value stores.

The same set of steps that lead to an over-count back in Fig. 6 are illustrated now with the key-value store using FBFs in Fig. 7. Steps $1-4$ are the same as in Fig. 6. However, each update has an unique id associated with it and each counter has an FBF integrated into them as shown in Fig. 7. After the loss of ack at the end of step 4, the operation with an id of *cli1_ts3* is retried. However, it is dismissed as a retry since it passes the membership test in the FBF. The correct value of the counter $c1$ (i.e., 6) is returned at the end of step 4.

In summary, using the FBF, it is possible to obtain idempotent (with high probabilistic accuracy), lock-free (without the lock-read-write-unlock design of Cassandra $v2.1$), correct, fast (due to the lock-free design), distributed counters in key-value stores such as Cassandra.

VI. Evaluation

We have implemented a standalone prototype of the FBF data structure in C++. We present experimental results that show its false positive rate, adaptability and the effect on counter operations.

To assist readers Table I summarizes the terms and notation used throughout this section (some of these terms have already been used in the paper, but this is a comprehensive summary).

TABLE I: Summary of Terms used in the experiments.

Notation	Definition
FPP	False Positive Probability
N	Number of constituent past Bloom filters in the FBF
m	Number of bits in each constituent Bloom filter of the FBF
k	Number of hash functions used by each constituent Bloom filter in the FBF
t	Refresh period of the FBF

A. Membership Check

We compare the false positive rate behaviors of the optimized membership check algorithm (Sec. III-D) against the simple approach of Sec. III-A (i.e., checking all constituent Bloom filters). This experiment was run with a basic FBF with the dynamic resizing thread disabled. The FBF parameters are $m = 6250$ bits, $k = 5$ hash functions, and a refresh period of $t = 5s$.

Fig. 8 plots the false positive probability (FPP) on the Y-axis against the number of elements inserted into the FBF on the X-axis. The membership check algorithms are initiated after all the elements are inserted. The FPP was computed empirically using both approaches by running a membership check for $200k$ invalid elements that were known not to be inserted in the FBF, and calculating what fraction of such checks returned true as an answer. As we can see from Fig. 8, the FPP using the optimized membership check algorithm (Sec. III-D) is much lower than using the simple approach (Sec. III-A) – notice that the vertical axis is logarithmic in scale. For instance, at 300 operations, the FPP computed using our optimized membership check is 90% lower than the FPP calculated using the simple approach.

We conclude that the optimized membership check is worth using as it provides higher accuracy and thus lowers the number of valid counter operations that would be dismissed incorrectly as retries.

B. False Positive Probability of an FBF

We now evaluate if our false positive analysis of Sec. IV in fact accurately models the observed FPP. The empirical FPP was determined by testing the FBF for membership against $200k$ invalid elements that were known not to be inserted into the FBF, and calculating the fraction that returned true as an answer. This is the ground truth.

The first experiment was run using a basic FBF (with the dynamic resizing thread disabled). The FBF parameters are $k = 5$ hash functions, $m = 6250$ bits, and a refresh period of $t = 5s$. Fig. 9 plots the FPP against the number of elements inserted into the FBF. The FPP determined using

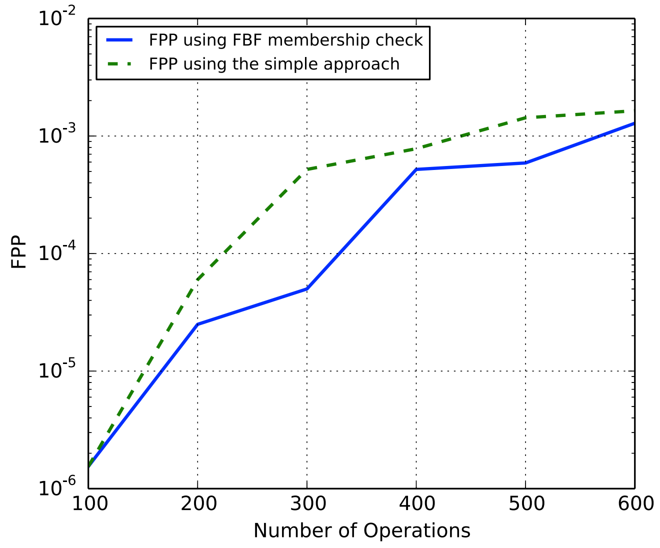


Fig. 8: False positive probability(FPP) using the optimized membership algorithm and the simple approach.

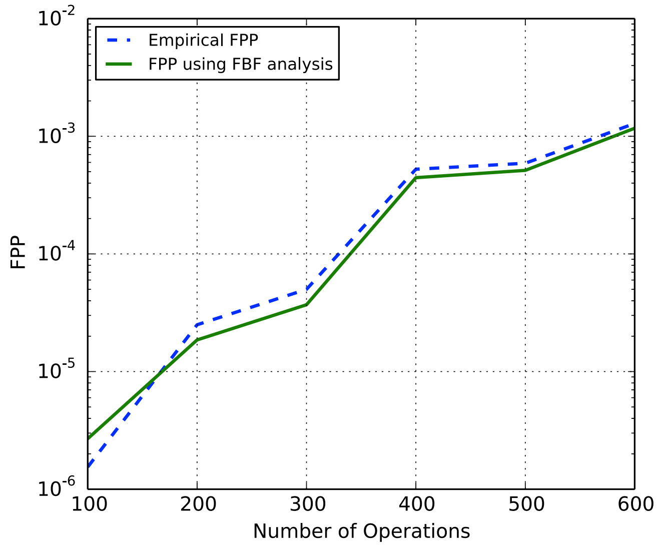


Fig. 9: False Positive Probability(FPP) of an FBF comparison with the empirical FPP.

our analysis reasonably matches the actual FPP determined empirically even as the number of elements inserted into the FBF increases.

A second experiment was run against multiple simple FBFs, each with different sizes of the constituent Bloom filters ranging from $5k$ to $30k$ bits, keeping the number of elements inserted into the FBF constant at 300 and $k = 5$ hash functions. Fig. 10 shows again that the analysis is reasonably accurate even if the size of each constituent Bloom filter in the FBF is varied.

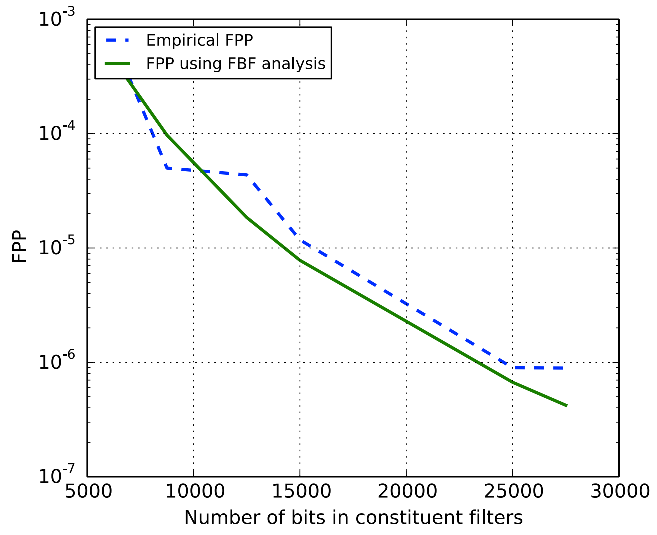


Fig. 10: False Positive Probability(FPP) of an FBF comparison with empirical FPP.

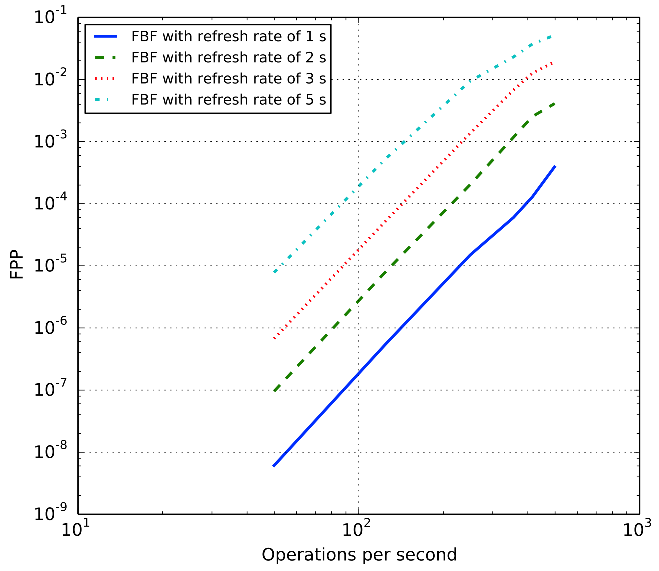


Fig. 11: False Positive Probability(FPP) of an FBF decreases as refresh period is decreased.

C. Dynamic Resizing

We separately evaluate the effect of the two adaptive knobs of Section III-E – 1) the refresh rate change, 2) the number of Bloom filters, and 3) evaluate the combined effect of adapting them together.

In the first experiment, 4 simple FBFs each with refresh period of $5s$, $3s$, $2s$ and $1s$ respectively are used. The FBF parameters are $m = 25k$ bits, and $k = 5$ hash functions. $5k$ elements were inserted with an exponential increase in the workload rate. As we can see from Fig. 11, the FPP decreases as the refresh rate is lowered. For instance, at 100 operations

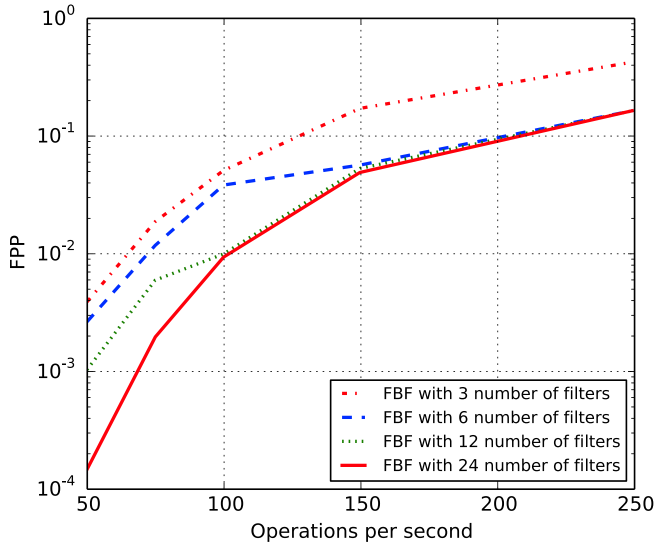


Fig. 12: False Positive Probability(FPP) of an FBF decreases as number of filters is increased.

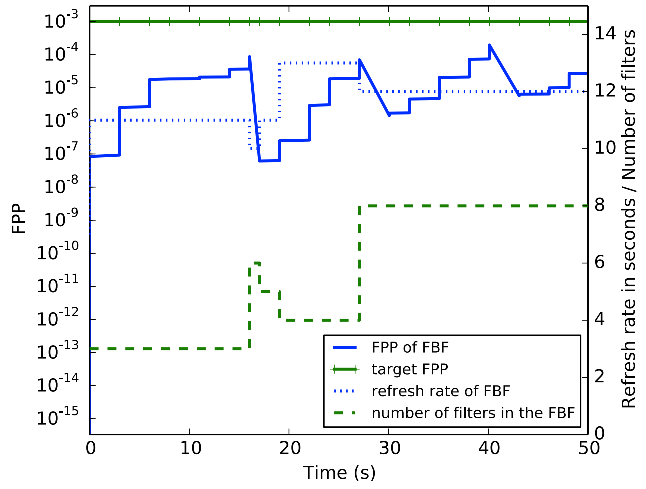


Fig. 13: False Positive Probability(FPP) of an FBF is maintained below an application provided target FPP even with increasing workload over time by dynamic resizing.

per second, the FPP of the FBF with a refresh period of $5s$ is 1000 times the FPP of the FBF with a refresh period of $1s$. This shows that refresh rate is an effective control knob for the FPP.

In the second experiment, 4 simple FBFs with 24, 12, 6 and 3 constituent Bloom filters are used. The FBF parameters are refresh period of $t = 5s$, $m = 6250$ bits, and $k = 5$ hash functions. $3k$ elements are inserted with a steady increase in the workload. As we can see from Fig. 12, the FPP of an FBF decreases as we scale up the number of filters in the FBF. For instance, at 100 operations per second, the FPP of the FBF with 3 Bloom filters is 5.5 times the FPP of the FBF with 24 Bloom filters. This is true even at high operation rates. We conclude that changing the number of constituent Bloom filters is an effective control knob for adaptation.

In the third experiment, we show that it is possible to maintain the FPP of an FBF below an application provided target FPP by adapting both the number of constituent Bloom filters as well as the refresh rate simultaneously. This is based on the algorithm described in Algorithm 3. Fig. 13 shows how the number of Bloom filters and refresh period are adapted to keep the overall FPP of the FBF below the target FPP. Initially, we begin with a simple FBF i.e., an FBF with 3 Bloom filters, each of which has $m = 6250$ bits and $k = 5$ hash functions and a refresh period of $11s$. The primary Y-Axis plots the FPP and the secondary Y-Axis plots the variation in the number of constituent Bloom filters and refresh period in seconds. As more elements are inserted, the FPP of the FBF increases. At $t = 16s$, when the FPP of the FBF comes within 10% of the target FPP, the dynamic resizing kicks in and the number of Bloom filters is scaled to 6 (i.e., $4 - FBF$) and refresh rate is decreased to $10s$. This brings the FPP of the FBF down. Later, as the workload starts to subside at $t = 19s$, the number of Bloom filters drops to 4 and the refresh period is at $13s$. Dynamic resizing kicks again at $t = 28s$ as the FPP of the FBF approaches the target FPP. It increases the number of constituent Bloom filters from 4 to 8 and the refresh period is decreased from $13s$ to $12s$. We can see the FPP of the FBF drops at $t = 40s$ as well. This is due to the periodic refresh operation based on the refresh period t.

We conclude that FBFs are capable of adapting to the varying workload of counter operations. It scales up and down transparently in order to preserve the accuracy of counter operations.

D. FBF integration with distributed key-value store

We built a C++ simulator capturing essential features of Cassandra-like distributed key-value (KV) store [27]. Our use of the simulator is motivated by our desire to show that FBFs are applicable to and useful in any distributed key-value store.

We set up a 10 server cluster on the Emulab testbed [28]. The nodes in the cluster are set up in a ring topology. Each node in the ring is a $d710$ node connected to its neighbor using a $100Mbps$ link. The node type information of a $d710$ node is shown in Table II.

TABLE II: Emulab d710 node type information.

OS	Ubuntu 14.04
Processor	64-bit Intel Quad Core Xeon E5530
Memory	2 GB

Each node is running the distributed key-value server. We create two counter objects each with a replication factor of 3. One of the counters has an FBF associated with it, while the other doesn't (these two counters are partitioned based on consistent hashing). Every update operation on the former counter goes through the FBF (i.e., a membership check is made using the unique id provided by the client in the FBF) and the counter is updated only if the id uniquely representing that update were not already present in the FBF. The latter counter object doesn't have an associated FBF. Update operations are directly applied on the object. Using a client, a set of identical counter increment operations are issued to both the counters. A subset of these counter operations are retried periodically. Fig. 14 plots the counter values along with the expected value against time. We observe that the expected value and the value of the counter with the FBF overlap at all times. At the end of the experiment, the counter with the FBF is 100% accurate, i.e., the value of the counter is number of counter increments without the retries. The counter without the FBF has an error of 6%. With write rates that are higher or vary over time, the FBF would self-adapt in order to keep the false positive rate low (as was shown in Figure 13).

Although our analysis showed there might be false positives, for all practical purposes, the FPP is so low that the counters are essentially accurate in our experiments.

Next, we fail (crash and take offline) one node among the nodes containing the counter with the FBF. The distributed key-value store automatically replicated the counter at another node. During this process, we observed that counters continued to provide 100% accuracy. This shows that FBF handles node failures and still provides accurate counters.

The counter objects with the FBFs integrated with them have an additional step of membership check that needs to be done in the FBF during every update operation. We ran another experiment to show that the effect on write latency due to this step is negligible. We set up four $d710$ nodes in a star topology and used another $d710$ node as a client. We then ran batch counter updates and measured write latency of counter update operations on both the counter with the FBF enabled and vanilla counters (i.e., counters without the FBF). Fig. 15 shows that we observed that counter with FBF for a batch counter updates of 50 operations is only slower by 10% which is in order of $2ms$ on average.

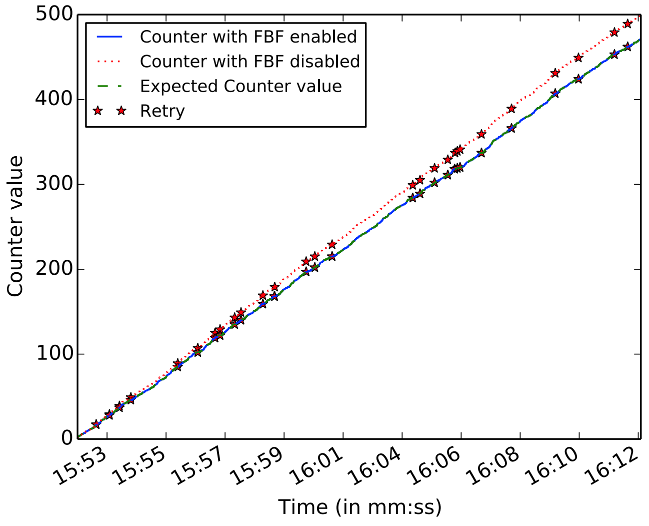


Fig. 14: Counter without FBF has an error of 6% as compared to counter with FBF, which is 100% accurate in our experiment.

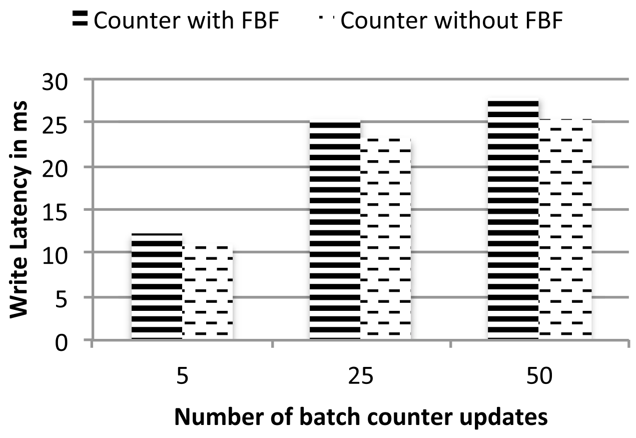


Fig. 15: Write latency in ms of batch counter updates.

VII. Related Work

The notion of remote procedure calls (RPCs) was formulated by Birrell and Nelson in [11]. The notion of at least/at most/exactly once semantics arises from these RPCs. It is well-known that exactly once semantics are hard to achieve in a distributed system with failures and message losses.

There are a variety of distributed key-value/NoSQL stores like Cassandra [2], RIAK [3], BigTable [4], DynamoDB [6], and Couchbase [29] among others. Most of them have support for counting using fast, distributed counters [7], [8]. Counters in most NoSQL stores are based on CRDTs [30].

Cassandra $v2.1$ contains a slightly more correct implementation of counters [31]. However it's new design has performance implications. The key difference from earlier versions is that instead of logging the deltas in the commitlog, making commitlog replays potentially unsafe, it reads the value of the counter for every update operation and then applies the delta. The new value is then updated in the Memtable. This makes the distributed counters slower since it involves lock-read-update instead of just adding the delta. The change solves internal idempotence i.e., makes the commitlog replays safe. However, it cannot solve external idempotence via replays from clients and can still lead to inaccuracies.

In contrast, our solution helps in achieving both internal (i.e., commitlog replay) as well as external (i.e., client replays) idempotence with high probabilistic accuracy thus providing correct counters. It also ensures that counter operations are lock-free and thus fast.

Another work [32] suggests making Apache Storm counters idempotent by using a combination of other systems like Redis [33] and Apache Kafka [34]. In contrast to this, our solution proposes an elegant design that can be integrated into a system without drastic changes in design or external dependency on other systems.

Bloom filters [13] have spawned a rich set of variants. Counting Bloom filter (CBF) [15] allows the set to change dynamically via insertions and deletions by maintaining a bit vector with a fixed cell width of w. Spectral Bloom Filters (SBF) [35] is designed primarily for multi-sets allowing estimates of multiplicities of keys with a small error probability. Stable Bloom filters [36] also maintains cells instead of bits and are used to eliminate duplicates in data streaming applications. After each insertion, randomly selected cells are decremented by 1 in order to make space for newer elements. However, none of these satisfy the requirements needed to maintain idempotent counters. It is not possible to automatically retire older operations and scale according to the workload.

VIII. Conclusion

Distributed key-value stores/NoSQL have become popular because they provide better scalability, availability and performance compared to traditional relational databases. In this paper, we introduced a novel data structure called *Forgetful Bloom Filter*, which we call an FBF. This data structure allows us to compactly maintain the recently received list of update operations, so that exactly-once semantics can be provided with high probability. We also presented techniques to adapt the FBF as the workload rate changes. We showed how to integrate the FBF into key-value stores like Cassandra. Our experimental evaluations show that the FBF can achieve low false positive probabilities, and achieve high accuracy of counter operations in key-value stores.

References

[1] A. Lakshman and P. Malik, "Cassandra: A decentralized structured storage system," *SIGOPS Operating Systems Review*, vol. 44, no. 2, pp. 35–40, Apr. 2010. [Online]. Available: http://doi.acm.org/10.1145/1773912.1773922

[2] "The Apache Cassandra Project." http://cassandra.apache.org/, Accessed: 2015-05-11.

[3] "RIAK." http://basho.com, Accessed: 2015-05-11.

[4] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," in *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 15–15. [Online]. Available: http://dl.acm.org/citation.cfm?id=1267308.1267323

[5] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's highly available key-value store," *SIGOPS Operating Systems Review*, vol. 41, no. 6, pp. 205–220, Oct. 2007. [Online]. Available: http://doi.acm.org/10.1145/1323293.1294281

[6] "AWS — Dynamo DB - NoSQL Cloud Database Service." http://aws.amazon.com/dynamodb, Accessed: 2015-05-11.

[7] "Using a counter." http://www.datastax.com/documentation/cql/3.0/cql/cql_using/use_counter_t.html, Accessed: 2015-05-11.

[8] "Counters in RIAK 1.4." http://basho.com/counters-in-riak-1-4, Accessed: 2015-05-11.

[9] "Databases — Research at Facebook." https://research.facebook.com/databases, accessed: 2015-05-11.

[10] "Rainbird: Real-time analytics at Twitter." http://cdn.oreillystatic.com/en/assets/1/event/55/Realtime%20Analytics%20at%20Twitter%20Presentation.pdf, Accessed: 2015-05-11.

[11] A. D. Birrell and B. J. Nelson, "Implementing remote procedure calls," *ACM Transactions on Computer Systems*, vol. 2, no. 1, pp. 39–59, Feb. 1984. [Online]. Available: http://doi.acm.org/10.1145/2080.357392

[12] "[CASSANDRA-2495] Add a proper retry mechanism for counters in case of failed requests." https://issues.apache.org/jira/browse/CASSANDRA-2495, Accessed: 2015-05-11.

[13] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, Jul. 1970. [Online]. Available: http://doi.acm.org/10.1145/362686.362692

[14] Wikipedia, "Bloom filter," 2015, [Online; accessed 11-May-2015]. [Online]. Available: http://en.wikipedia.org/wiki/Bloom_filter

[15] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: A scalable wide-area web cache sharing protocol," *IEEE/ACM Transactions on Networks*, vol. 8, no. 3, pp. 281–293, Jun. 2000. [Online]. Available: http://dx.doi.org/10.1109/90.851975

[16] V. Jacobson, "Congestion avoidance and control," in *Symposium Proceedings on Communications Architectures and Protocols*, ser. SIGCOMM '88. New York, NY, USA: ACM, 1988, pp. 314–329. [Online]. Available: http://doi.acm.org/10.1145/52324.52356

[17] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet mathematics*, vol. 1, no. 4, pp. 485–509, 2004.

[18] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," in *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, ser. STOC '97. New York, NY, USA: ACM, 1997, pp. 654–663. [Online]. Available: http://doi.acm.org/10.1145/258533.258660

[19] "Counters in Cassandra." http://wiki.apache.org/cassandra/Counters, Accessed: 2015-05-11.

[20] "Durability Cassandra." http://wiki.apache.org/cassandra/Durability, Accessed: 2014-05-11.

[21] "MemTable in Cassandra." http://wiki.apache.org/cassandra/MemtableSSTable, Accessed: 2015-05-11.

[22] "Partitioned Counters Design Document." https://issues.apache.org/jira/secure/attachment/12459754/Partitionedcountersdesigndoc.pdf, Accessed: 2015-05-11.

[23] "Read Repair on Apache Cassandra Wiki." http://wiki.apache.org/cassandra/ReadRepair, Accessed: 2015-05-11.

[24] "Anti-Entropy on Apache Cassandra Wiki." https://wiki.apache.org/cassandra/AntiEntropy, Accessed: 2015-05-11.

[25] W. Vogels, "Eventually consistent," *Queue*, vol. 6, no. 6, pp. 14–19, Oct. 2008. [Online]. Available: http://doi.acm.org/10.1145/1466443.1466448

[26] "SSTable in Cassandra." http://wiki.apache.org/cassandra/MemtableSSTable, Accessed: 2015-12-16.

[27] "A C++ Cassandra Simulator." https://github.com/rajath26/CassandraSimulator, Accessed: 2015-05-11.

[28] "Network emulation testbed home." https://www.emulab.net/, accessed: 2015-05-11.

[29] "Couchbase." http://www.couchbase.com/, Accessed: 2015-05-11.

[30] M. Shapiro, N. Preguiça, C. Baquero, and M. Zawirski, "Conflict-free replicated data types," in *Proceedings of the 13th International Conference on Stabilization, Safety, and Security of Distributed Systems*, ser. SSS'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 386–400. [Online]. Available: http://dl.acm.org/citation.cfm?id=2050613.2050642

[31] "What's new in Cassandra 2.1: Better Implementation of Counters." http://www.datastax.com/dev/blog/whats-new-in-cassandra-2-1-a-better-implementation-of-counters, Accessed : 2014-12-16.

[32] "No more Over-Counting: Making Apache Storm Counters Easy and Idempotent using Kafka and Redis." https://blog.deck36.de/no-more-over-counting-\ \making-counters-in-apache-storm-idempotent-using-redis-hyperloglog/, Accessed: 2015-05-11.

[33] "Redis." http://redis.io/, Accessed: 2015-05-11.

[34] "Apache Kafka." http://kafka.apache.org/, Accessed: 2015-05-11.

[35] S. Cohen and Y. Matias, "Spectral Bloom Filters," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '03. New York, NY, USA: ACM, 2003, pp. 241–252. [Online]. Available: http://doi.acm.org/10.1145/872757.872787

[36] F. Deng and D. Rafiei, "Approximately detecting duplicates for streaming data using stable bloom filters," in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '06. New York, NY, USA: ACM, 2006, pp. 25–36. [Online]. Available: http://doi.acm.org/10.1145/1142473.1142477