CRDTs for Approximate Membership Queries

Junbo Xiong Trovato
j.xiong-6@student.tudelft.nl
Delft University of Technology
Delft, The Netherlands

Ege Berkay Gulcan
e.b.gulcan@tudelft.nl
Delft University of Technology
Delft, The Netherlands

Burcu Kulahcioglu Ozkan
b.ozkan@tudelft.nl
Delft University of Technology
Delft, The Netherlands

Abstract

Probabilistic data structures are used in applications that manage large datasets due to their time and space efficiency. These applications can accommodate approximate results from probabilistic data structures and replicated systems that use them can take advantage of the efficiency gained from weaker synchronization and consistency among replicas.

In this paper, we propose conflict-free replicated data types (CRDTs) for probabilistic data structures with approximate membership queries. Specifically, we introduce Conflict-Free Replicated Bloom Filters and Conflict-Free Replicated Cuckoo Filters, which are the conflict-free versions of traditional Bloom and Cuckoo filters. We provide implementations of these data structures in an open-source repository and present an evaluation of the approximate query results across various workload and replica synchronization configurations.

CCS Concepts: • **Theory of computation → Distributed algorithms**; • **Information systems → Distributed storage**; • **Software and its engineering → Consistency**.

Keywords: conflict-free replicated data types, replica consistency, probabilistic data structures, approximate membership

1 Introduction

Probabilistic data structures are fast and space-efficient data structures that have gained popularity for big data applications with large data sets [11]. Unlike regular data structures, probabilistic data structures return approximated answers with some estimation of possible errors. For instance, a probabilistic set can offer approximate membership queries, returning either "possibly in the set" or "definitely not in the set" when checking if an element is contained within it, with certain bounds on the false positive rate.

These structures offer an efficient solution for applications that keep vast amounts of data, need fast processing time, and can tolerate approximate answers. One of the most popular examples of such data structures, Bloom filters [4], are used in many systems to identify malicious URLs, used in routing tables [28], prefix matching of IP addresses [8] and efficient network state management [24].

Besides allowing for approximate answers, many use cases for probabilistic data structures often do not demand strong synchronization of their copies. Consider an example replicated system that maintains an approximate membership data structure of blocklisted email addresses to identify malicious addresses. Such a system can reduce synchronization costs by permitting concurrent updates across different copies of the probabilistic data structure instead of synchronizing all copies after each element is inserted.

Conflict-free Replicated Data Types (CRDTs) [23] are well-known data structures that support concurrent update operations. CRDTs are equipped with conflict resolution mechanisms that provide automated merging of the concurrent updates without involving global coordination and guarantee convergence of the replica states. CRDTs have been designed for many data structures including registers [23], counters [23], sets [3, 23], maps [7, 21], lists [20, 23, 26], graphs [23], JSON documents [15], priority queues [29].

In this work, we propose CRDTs for probabilistic data structures with approximate membership queries. We present two new CRDTs for grow-only probabilistic data structures: the Conflict-free Replicated Bloom Filter (CRBF) and grow-only Conflict-free Replicated Cuckoo Filter (CRCF), which are the conflict-free versions of the Bloom filter [4] and Cuckoo filter [9]. We provide their specifications following the state-based approach for replicated data types [23] and formulate the error estimation of their membership queries based on the error estimation of their regular versions.

We verified the correctness of our CRDT specifications using VeriFx [19] and provided their implementations in an open-source library [25]. Our evaluation of CRBFs and CRCFs on different workload and replica synchronization configurations shows that the empirical false positive rates of the conflict-free versions of the data structures align well with their theoretically estimated values and do not significantly differ from their regular versions, especially when the replicas synchronize frequently.

2 Data Structures for Approximate Queries

2.1 Bloom Filter

Bloom filter [4] is the most well-known data structure for approximate membership queries. It consists of a bit array

PaPoC '25, March 30–April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1558-7/25/03
https://doi.org/10.1145/3721473.3722146

```
   Input: Filter B with k hash functions {h_1, ..., h_k}
   Input: Element e ∈ D
1  proc add(B, e)
2      for i ∈ {1, ..., k} do B[h_i(e)] ← 1
3  proc contains(B, e)
4      for i ∈ {1, ..., k} do
5          if B[h_i(e)] ≠ 1 then return false
6      return true
```

Algorithm 1: Adding an element and querying the membership of an element in a Bloom filter.

of m bits and k independent hash functions $(h_1, h_2, \ldots, h_k)$. Initially, all the indices of the bit array are set to 0.

A Bloom filter supports operations for adding an element and querying the membership of an element. Algorithm 1 provides the procedures for these operations. An element e is added to the filter by computing its hash using all the hash functions $(h_1, h_2, \ldots, h_k)$ to obtain k index values with $\{h_i(e) \bmod m\}_{i=1}^{k}$ and setting these indices of the filter to 1. Querying the membership of an element checks the indices generated by the hash functions. If all the indices generated by the hash functions are set to 1, then the element is *possibly* contained in the set. Otherwise, it is not contained in the set.

As the bit at an index may be set by multiple elements due to hash collisions, a Bloom filter can return *false positive* membership results. The probability of getting a false positive membership is given by Equation 1, where n is the expected number of elements in the filter [4].

$$\varepsilon = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^{k} \approx (1 - e^{-kn/m})^{k} \tag{1}$$

2.2 Cuckoo Filter

Cuckoo filter [9] is an alternative to Bloom filter that uses Cuckoo hash tables [18] with an optimization called partial-key cuckoo hashing, which allows storing only a portion of the key, i.e., its fingerprint; instead of the entire key. The Cuckoo filter improves the Bloom filter by providing better false positive rates while maintaining space efficiency and allowing the removal of elements. In this paper, we present a conflict-free data type for the grow-only Cuckoo filter and provide element insertion and querying operations.

A Cuckoo filter consists of a Cuckoo hash table with m buckets each storing up to b values, a fingerprint function h_F, and a hash function h. Algorithm 2 provides the addition and membership checking of an element. An element e is inserted into the filter by computing its fingerprint $f \leftarrow h_F(e)$, and its two candidate buckets $i \leftarrow h(e)$ and $j \leftarrow i \oplus h(f)$, where $\oplus$ denotes the bit-wise XOR operation. The element is inserted into the bucket at i if it is not full (line 4) or the alternative bucket at j if not full (line 6). If both are full, it randomly chooses one of those buckets (line 7) to store e there. It displaces the element from that bucket to its alternative candidate bucket, which can be computed without using the

```
   Input: Filter C with fingerprinting h_F and hash function h
   Input: Element e ∈ D
1  proc add(C, e)
2      f ← h_F(e), i ← h(e), j ← i ⊕ h(f)
3      if C[i] has empty space then
4          C[i].add(f) return true
5      else if C[j] has empty space then
6          C[j].add(f) return true
7      k ← randomChoose({i, j})
8      for i ∈ {0, ..., MaxIter} do
9          x ← randomChoose(C[k])
           // swap f and the fingerprint stored in x
10         k ← k ⊕ h(f)
11         if C[k] has empty space then
12             C[k].add(f) return true
13     return false
14 proc contains(C, e)
15     f ← h_F(e), i ← h(e), j ← i ⊕ h(f)
16     if f ∈ C[i] or f ∈ C[j] then return true
17     return false
```

Algorithm 2: Adding an element and querying the membership of an element in a Cuckoo filter.

original element but only the stored fingerprint (line 11). The displacement repeats until an empty slot (lines 12-13) or a maximum number of iterations is reached (line 8). If there are no empty buckets, the insertion fails (line 14).

The membership query in a Cuckoo filter is similar to the query of a Bloom filter. A positive result indicates the existence of an entry with fingerprint f in either bucket i or j. Like Bloom filters, Cuckoo filters suffer from false positives, with the probability given in Equation 2, where l is the length of a fingerprint in bits, and c is the bucket size.

$$\varepsilon = 1 - \left(1 - \frac{1}{2^l}\right)^{2c} \approx \frac{2c}{2^l} \tag{2}$$

3 Conflict-free Data Structures for Approximate Membership

We present two CRDTs for probabilistic data structures for approximate membership queries: Conflict-free Replicated Bloom Filter (CRBF) and Conflict-free Replicated Cuckoo Filter (CRCF), following the state-based CRDT approach [22].

CRDT Specifications. The specifications define the state representation and the initial state of the CRDT, query operations that do not modify the state, and modification operations that update the local state of the source replica, where the updated state is broadcasted to other replicas, asynchronously. For CRBF and CRCF, the query and modification operations are, respectively, *contains*, which checks approximate membership, and *add*, which adds an element.

For the convergence of replica states, the specification also defines the functions *compare* and *merge*. The *compare* function defines a partial order on the replica states, which returns `true` if the state of the current replica is partially less than or equal to `other` and `false` otherwise. The function *merge* yields the least upper bound of two states following the

partial order. In practice, *merge* is called when a message is received to merge the received state with the replica's current state. State-based CRDTs guarantee convergence if the merge function is idempotent, commutative, and associative [22].

Overview of CRBF and CRCF. Our specifications represent the filter states using grow-only sets and use grow-only set CRDTs [22]. This representation of the CRBF states offers query, add, and merge operations following the set CRDTs.

The specification of CRCF is more involved with additional challenges on the element insertion and state merging. Unlike a Bloom filter, which always allows for the addition of elements, a Cuckoo filter rejects the addition of an element if its corresponding bucket is full. This restriction creates difficulties in replicated systems where updates are non-coordinated. In such scenarios, two concurrent addition operations may both succeed, even though only one would succeed in a sequential setting. One approach to address this challenge is to introduce reservations or implement stronger synchronization for certain operations [1, 2, 17] which are used to ensure critical safety guarantees. However, since the bucket specifications of probabilistic data types are not safety-critical but are mainly used for error estimation, we adopt an alternative strategy. We relax the restriction on bucket size, allowing an element to be added as long as the corresponding bucket in the local replica is not full. Moreover, we retain all elements added in the replicas after the merge operation. We mitigate the effects of overflowing buckets during element insertion, as explained in Section 3.2.

Another challenge arises from state merging of the Cuckoo filters, as a naïve approach that is suitable for merging the set-based representations of Bloom filters could lead to redundant entries in Cuckoo filters. This redundancy increases the load factor of the Cuckoo filter, resulting in higher false positive rates. To address this issue, we propose a more sophisticated merging method that takes into account the alternate buckets of entries in the Cuckoo table while merging two filters and, thus, prevents duplicate entries.

3.1 Conflict-free Replicated Bloom Filter (CRBF)

CRBF represents a Bloom filter that consists of an m-bit bit array with a grow-only set by modeling the bit array as a set of active indexes. The possible states of a CRBF are partially ordered by the set inclusion relation. The initial state of a CRBF is an empty set as all its bits are initially set to 0.

Data Type Operations. Figure 3 provides the specification of the CRBF. The query and update operations of the CRBF follow from the operations of the Bloom filter. Querying the membership of an element computes a set of indices by hashing e using the functions $\{h_i(e)\}_{i=1}^{k}$. If the computed set of indices is the subset of the current state of the filter, then e is considered to be contained by the filter. Similarly, adding an element e to the filter updates its state by taking

payload S: set⟨index⟩
initial $S \leftarrow \emptyset$
query *contains(e)*:
  return $\{h_i(e)\}_{i=1}^{k} \subseteq S$
update *add(e)*:
  return $\{h_i(e)\}_{i=1}^{k} \cup S$
compare (S'):
  return $S \subseteq S'$
merge (S'):
  return $S \cup S'$

Specification 3: Conflict-free Bloom filter

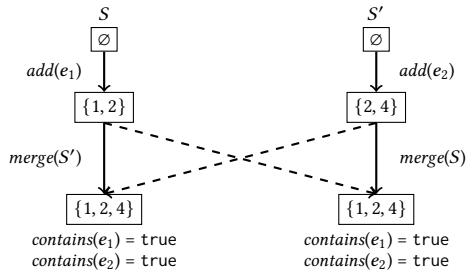


Figure 1. An example of merging two Bloom filter states.

the union of the current filter state and the set of indices computed by hashing e using the hash functions.

State Convergence. The states of two CRBFs are merged simply by taking the set union of their states. Figure 1 illustrates an example of adding two elements e_1 and e_2 into two CRBFs with two hash functions. In the example, e_1 is hashed to $\{1, 2\}$ and e_2 to $\{2, 4\}$. The merged state contains all active indices introduced by e_1 and e_2. Hence, the membership queries for both elements yield `true`.

Probabilistic Characteristics. The false positive rate of the CRBFs does not differ from the regular Bloom filters given in Equation 1, where m is the number of bits, k is the number of hash functions, and n is the total number of elements added to the replicated Bloom filter.

3.2 Conflict-Free Replicated Cuckoo Filter (CRCF)

Similar to the regular Cuckoo filter, the CRCF holds a Cuckoo hash table that we represent as a set of pairs of integers in the specification. The pair (i, f) denotes an entry of fingerprint f in the i-th bucket. The algorithms for calculating the fingerprint and bucket indexes are all identical to the description in the Cuckoo filter algorithm in Section 2.2.

A major difference between CRCF and the regular Cuckoo filters is that while the capacity c of a bucket is a hard limit in the standard cuckoo filter, it becomes a soft one for the CRCF. This is because CRCF allows the replicas to add entries if their buckets are not full and keeps all inserted entries in the filter after a merge operation. We consider a bucket *full* if it contains exactly c entries *full* and *overflowing* if more entries are inserted.

Data Type Operations. Specification 4 describes the operations of a CRCF. The membership query follows the regular Cuckoo filters, i.e., it returns true if either of the pairs (i, f)

payload S: set⟨index × fingerprint⟩
initial $S \leftarrow \emptyset$
query *contains(e)*:
  $f \leftarrow h_F(e), i \leftarrow h_I(e), j \leftarrow altIndexOf(i, f)$
  return $(i, f) \in S \ \vee \ (j, f) \in S$
update *add(e)*:
  if *contains(e)* **then return** S
  return *cuckooInsert*(S, e)
compare *(S′)*:
  $D \leftarrow \{(altIndexOf(i, f), f) \mid (i, f) \in S\}$ // dual
  $D' \leftarrow \{(altIndexOf(i, f), f) \mid (i, f) \in S'\}$
  return $(S \cup D) \subseteq (S' \cup D')$
merge *(S′)*:
  $D \leftarrow \{(altIndexOf(i, f), f) \mid (i, f) \in S\}$
  return $S \cup (S' \setminus D)$

Specification 4: Conflict-free Cuckoo Filter

and (j, f) is contained in the set, where f is the fingerprint, i is the bucket index, and $j = altIndexOf(i, f)$ is the alternative bucket index.

As CRCF relaxes the Cuckoo filter's bucket capacity constraint, we extend the procedure of adding elements with a balancing function. Algorithm 5 shows the procedure for inserting an element e into a CRCF with state S. Similar to the regular Cuckoo filters, it checks both candidate buckets for available slots (line 3). If only one of the buckets has a slot, it inserts the entry in that bucket (lines 4-5). If both buckets are available, it chooses a random bucket to insert the element (lines 6-7). The steps in *cuckooInsertImpl* move entries to their alternative buckets to accommodate the newly inserted one, up to $maxIters$ times. If the current bucket is not full, the entry f is inserted in that bucket (lines 12-13). If it is full, the entry replaces a random one in the bucket (lines 14-16), and then the victim entry f' is inserted into its alternative bucket (lines 17-18). If the bucket is overflowing, the algorithm first tries "smoothing out" the excess entries by moving them (lines 19-23). Then, the remaining quota of iterations are spent on inserting the original entry f (line 24). The procedure *cuckooInsertImpl* aborts if it exceeds the maximum number of attempts as in a regular Cuckoo filter. It ensures no additional overflowing bucket is introduced due to local *adds* and redistributes the entries on the fly to rebalance the cuckoo hash table.

State Convergence. A straightforward approach to define the *merge* operation is directly using set union: $S \sqcup S' = S \cup S'$. While being theoretically correct, the naïve solution leads to duplicate entries corresponding to the same element.

Figure 2 demonstrates such an example with two replica states S and S'. Consider the elements e_1, e_2, e_3 having candidate buckets $\{1, 9\}$, $\{2, 8\}$, and $\{3, 7\}$, respectively. One of the replicas inserts e_1 and e_2, and another one inserts e_1 and e_3, possibly resulting in the replica states in Figure 2a. The naïve merge operation takes the union of all (*index*, *fingerprint*) pairs, which keeps duplicated entries for some elements. Duplicate entries fill in more buckets, increasing the filter's load factor and, hence, false positive rates.

1 **proc** *cuckooInsert(S, e)*
2 $f \leftarrow h_F(e), i \leftarrow h_I(e), j \leftarrow altIndexOf(i, f)$
3 $B_1 \leftarrow \{(i, f') \in S\}, B_2 \leftarrow \{(j, f') \in S\}$
4 **if** $|B_1| < c \ \wedge \ |B_2| \geq c$ **then** $index \leftarrow i$
5 **else if** $|B_2| < c \ \wedge \ |B_1| \geq c$ **then** $index \leftarrow i_2$
6 **else** $index \leftarrow randomChoose(\{i, j\})$
7 $(S', \_) \leftarrow cuckooInsertImpl(S, index, f, maxIters)$
8 **return** S'
9 **proc** *cuckooInsertImpl(S, i, f, n)*
10 **if** $n < 0$ **then throw** *"max #iterations is reached"*
11 $B \leftarrow \{f' \mid (i, f') \in S\}$
12 **if** $|B| < c$ **then** // Insert into an available slot
13 **return** $(S \cup \{(i, f)\}, n)$
14 **else if** $|B| = c$ **then** // Move an entry to its alt. bucket
15 $f' \leftarrow randomChoose(B)$
16 $S' \leftarrow S \setminus \{(i, f')\} \cup \{(i, f)\}$
17 $j \leftarrow altIndexOf(i, f')$
18 **return** $cuckooInsertImpl(S', j, f', n - 1)$
19 **else** // Evict excess entries in the bucket
20 $f' \leftarrow randomChoose(B)$
21 $S' \leftarrow S \setminus \{(i, f')\}$
22 $j \leftarrow altIndexOf(i, f')$
23 $(S'', n'') \leftarrow cuckooInsertImpl(S', j, f', n - 1)$
24 **return** $cuckooInsertImpl(S'', i, f, n'')$

Algorithm 5: Inserting an element e into a CRCF

We define a merge operation that eliminates duplicate entries while guaranteeing convergence. The operation uses the fact that in a Cuckoo table, the same fingerprint appearing in either candidate bucket indicates the existence of the same element (ignoring hash collisions). That is, a Cuckoo hash table state S conceptually holds a larger set of entries. This conceptual set U comprises both the entries included in S and equivalent entries in their alternative buckets. We call the latter the *dual Cuckoo hash table* D of S, given by:

$$D(S) = \{(altIndexOf(i, f), f) \mid (i, f) \in S\}. \tag{3}$$

$$\text{Then, we have:} \qquad U(S) = S \ \cup \ D(S). \tag{4}$$

Therefore, the partial order of two CRCF states can be defined based on the inclusion relation of their universe sets:

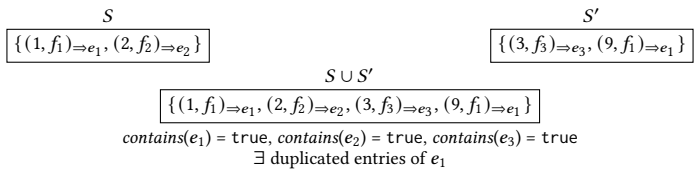
$$S \leq S' \Leftrightarrow U(S) \ \subseteq \ U(S'). \tag{5}$$

Since a Cuckoo table conceptually includes the elements in its dual table, when merging states, it only needs to retain entries that are not covered by its dual table. Accordingly, we define the *merge* function as:

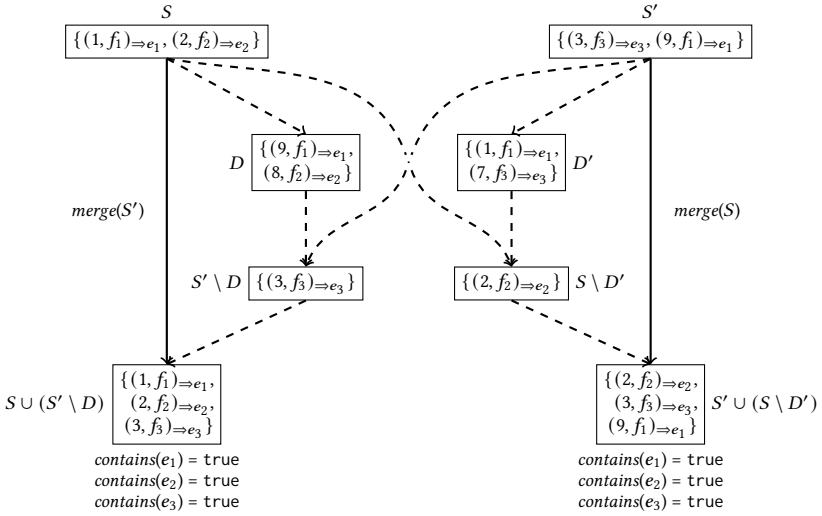
$$S \sqcup S' = S \ \cup \ (S' \setminus D(S)). \tag{6}$$

Figure 2b illustrates the merge function. The replica with state S merges with S' by first computing its dual Cuckoo table D and then inserting only the pairs from S' into S that are not contained in D. The replicas reach an equivalent state after they merge their states. The merged states contain all three elements, avoiding duplication.

Probabilistic Characteristics. The false positive estimation of CRCF follows from that of regular Cuckoo filters in Equation 2, with a difference that the load factor of some buckets in CRCF may exceed 100%. Therefore, we incorporate the average load factor, $\bar{\alpha}$, into the formula. While in the



(a) The result of *merge* using plain set union of states contains duplicated entries.



(b) The detailed procedure of *merge* using Equation 6. The merged states do not contain duplicated entries.

Figure 2. An example of merging two CRCFs using different merge operations. The fingerprint of element e_i is denoted as f_i. The indexes of candidate buckets are $\{1, 9\}$ for e_1, $\{2, 8\}$ for e_2, and $\{3, 7\}$ for e_3. The subscripts of the entries show the elements they correspond to. All merged states are equivalent, containing all three elements. While the method in Figure 2a results in duplicate entries for an element, the method in Figure 2b eliminates duplication.

regular filters, at most $2c$ entries are compared with a given fingerprint, in CRCF, $2c\bar{\alpha}$ slots are accessed per query on average. Hence, the probability of false positives becomes:

$$\varepsilon = 1 - \left(1 - \frac{1}{2^l}\right)^{2c\bar{\alpha}} \approx \frac{2c\bar{\alpha}}{2^l} \tag{7}$$

3.3 Verification

We verified the specifications of CRBF and CRCF using VeriFx [19], which offers a Scala-like high-level programming language for writing CRDT specifications and automatically verifying them. For simplicity of the verification, we use elementary hash functions in the verified specifications. Note that this does not affect the correctness verification of the data structures since the correctness of the Bloom and Cuckoo filters does not depend on the used hash functions. The analysis shows that both CRBF and CRCF achieve the commutative, idempotent, and associative properties, verifying the correctness of our specifications. We provide the VeriFx specifications in our GitHub repository [25].

4 Implementation and Evaluation

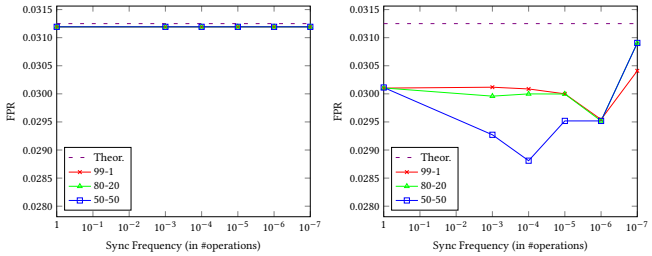
4.1 Implementation

We implemented the CRDTs for approximate membership queries as a Scala library of data structures and made it publicly available [25]. Our implementation uses `MurmurHash3` and `FarmHash` provided by the Google Guava [12] due to their performance and low collision rate. For CRBF, these two hash functions serve as the foundation for generating a series of hash codes, following the optimization approach by Kirsch and Mitzenmacher [14]. For CRCF, `MurmurHash3` is used for index hashing, and `FarmHash` for fingerprinting.

4.2 Evaluation

We evaluated the CRBF and CRCF in the (i) empirical false positive rates of their membership queries under different workload distributions and synchronization frequencies and (ii) space efficiency compared to regular set CRDTs in Akka [16].

Experimental Setup. We generated the workloads using a similar setup to previous works [6, 9, 13] and used 128-bit random integers generated by the standard library of Java. We created filters capacity matching the quantity of workload, used five hash functions for the Bloom filters, and



(a) CRBF **(b)** CRCF

Figure 3. False positive rates of the data structures under different workload and synchronization configurations

used a bucket of size 4 entries with an 8-bit fingerprint for the Cuckoo filters. Cuckoo filters are all configured with an iteration quota of 500 attempts, following the evaluation setup in the original work [9]. We distributed the workload among two replicas. We use d-$(100-d)$ *split* to label the distribution scheme where one replica conducts d% of the workload while the other does the rest.

False Positive Rate (FPR). We measured the empirical FPR of the membership queries invoking 2^{20} *add* operations split among two replicas according to a particular workload distribution and merging them every 10^3 to 10^7 operations, with an additional merge at the end of the execution. We used three workload distribution schemes with 50-50, 80-20, and 99-1 splits. For reference, we also evaluated the FPR of fully synchronized replicas, which corresponds to the regular versions of the structures. We computed the FPR by testing 2^{20} fresh random keys that have not been inserted and checking if they are reported as being contained. We repeated the experiments five times using different keys and testing datasets and reported the average values.

Figure 3a and Figure 3b show the FPR of CRBF and CRCF. For CRBF, the FPR is identical to the regular Bloom filter and is close to the theoretical value. The value is unaffected by the distribution scheme or synchronization frequency. This is because the merged state is always identical to the state resulting from local *adds*. For CRCF, the FPR varies across configurations. When the filters are merged relatively frequently, the FPR drops as the distribution becomes more even. This can be explained by the counterbalancing behaviors of *add* and *merge*. *merge* is inclined to introduce overflowing buckets to accommodate all the entries in the merged filters. Conversely, *add* does not introduce overflowing buckets and acts to balance them. In case of an even split, both replicas have many full buckets before merging. This leads to more overflowing buckets in the merged state, making the subsequent *adds* more likely to fail. The filter thus reports a lower load factor and, therefore, a lower FPR. However, if the filters are infrequently merged, the relation between those factors inverts. This is due to the uptick in the load factor.

Table 1. Serialized sizes of the CRBF, CRCF, and GSet CRDTs. BPE stands for *bytes per element*.

	CRDT	Load Factor	Serialised Size (MB)	Serialised BPE	Compressed Size (MB)	Compressed BPE
all local	CRBF	100%	1.01	1.01	**0.91**	0.91
	CRCF	96%	1.05	1.05	**1.04**	1.04
	GSet	100%	23.07	22.00	17.16	16.37
50-50 split	CRBF	100%	1.01	1.01	**0.91**	0.91
	CRCF	98%	3.73	3.62	**1.59**	1.54
	GSet	100%	23.07	22.00	17.16	16.37

If the workload is split into two replicas evenly, the merged filters reach a higher load factor, resulting in a higher FPR. Consequently, we observe that FPR first decreases and then increases as the synchronization frequency decreases. Yet, all observed FPR values remain below the theoretical estimates.

Space Efficiency. We evaluated the space efficiency of the CRBF and CRC compared to keeping the entries in regular `GSet` CRDT implementation in Akka [16]. We compared their serialized sizes, i.e., the number of bytes for network transmission. We measured and reported their raw sizes and GZIP compressed sizes, commonly used for web protocols.

Table 1 shows the serialized sizes of CRBF, CRCF, and `GSet`. The results show that CRBF and CRCF introduce substantial space efficiency compared to `GSet` while maintaining a high load factor. The size of the CRBF and GSet does not change across different workload split configurations as they allow inserting all elements, and they reach the same states in both configurations. On the other hand, we observe higher sizes of CRCF in 50-50 split due to the overflowing buckets.

5 Related Work

Recent work utilizes probabilistic data structures in the implementation of CRDTs. The work in [5] introduces Bloom-CRDT, an OR-Set CRDT that provides the semantics of the ORSet probabilistically. Unlike ORSet, which uses a set to record tombstones, BloomCRDT uses a Bloom filter as an approximation. Recent work [10] explores the use of Age-Partitioned Bloom Filters (APBFs) in set CRDTs. They show that using APBFs for CRDT causal context offers improvements in efficiency and memory management. In this work, we define CRDT versions of the probabilistic data structures.

6 Conclusion and Future Work

This paper introduced conflict-free replicated probabilistic data types for approximate membership queries in replicated systems: Conflict-free Replicated Bloom Filter and Conflict-free Replicated Cuckoo Filter. We provided the specification and verification of the CRDTs along with their implementation in an open-source library implementation.

Our ongoing work extends the library of conflict-free probabilistic data structures with more structures with automated scaling [27], approximate counting, and frequency queries.

Acknowledgements

We would like to thank Kevin De Porre and Elisa Gonzalez Boix for helpful discussions on using VeriFx.

References

[1] Valter Balegas, Sérgio Duarte, Carla Ferreira, Rodrigo Rodrigues, Nuno M. Preguiça, Mahsa Najafzadeh, and Marc Shapiro. 2015. Towards Fast Invariant Preservation in Geo-replicated Systems. *ACM SIGOPS Oper. Syst. Rev.* 49, 1 (2015), 121–125. https://doi.org/10.1145/2723872.2723889

[2] Valter Balegas, Cheng Li, Mahsa Najafzadeh, Daniel Porto, Allen Clement, Sérgio Duarte, Carla Ferreira, Johannes Gehrke, João Leitão, Nuno M. Preguiça, Rodrigo Rodrigues, Marc Shapiro, and Viktor Vafeiadis. 2016. Geo-Replication: Fast If Possible, Consistent If Necessary. *IEEE Data Eng. Bull.* 39, 1 (2016), 81–92. http://sites.computer.org/debull/A16mar/p81.pdf

[3] Annette Bieniusa, Marek Zawirski, Nuno M. Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. 2012. An optimized conflict-free replicated set. *CoRR* abs/1210.3368 (2012). arXiv:1210.3368 http://arxiv.org/abs/1210.3368

[4] Burton H. Bloom. 1970. Space/Time Trade-offs in Hash Coding with Allowable Errors. *Commun. ACM* 13, 7 (1970), 422–426. https://doi.org/10.1145/362686.362692

[5] Ewout Bongers. 2021. *Conflict Free R Tree: A CRDT-based index for P2P systems.* Master's Thesis. Delft University of Technology.

[6] Alexander Dodd Breslow and Nuwan Jayasena. 2018. Morton Filters: Faster, Space-Efficient Cuckoo Filters via Biasing, Compression, and Decoupled Logical Sparsity. *Proc. VLDB Endow.* 11, 9 (2018), 1041–1055. https://doi.org/10.14778/3213880.3213884

[7] Russell Brown, Sean Cribbs, Christopher Meiklejohn, and Sam Elliott. 2014. Riak DT map: a composable, convergent replicated dictionary. In *Proceedings of the First Workshop on the Principles and Practice of Eventual Consistency, PaPEC@EuroSys 2014, April 13, 2014, Amsterdam, The Netherlands*, Marc Shapiro (Ed.). ACM, 1:1. https://doi.org/10.1145/2596631.2596633

[8] Sarang Dharmapurikar, Praveen Krishnamurthy, and David E. Taylor. 2006. Longest prefix matching using bloom filters. *IEEE/ACM Trans. Netw.* 14, 2 (2006), 397–409. https://doi.org/10.1145/1217619.1217632

[9] Bin Fan, David G. Andersen, Michael Kaminsky, and Michael Mitzenmacher. 2014. Cuckoo Filter: Practically Better Than Bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies, CoNEXT 2014, Sydney, Australia, December 2-5, 2014*, Aruna Seneviratne, Christophe Diot, Jim Kurose, Augustin Chaintreau, and Luigi Rizzo (Eds.). ACM, 75–88. https://doi.org/10.1145/2674005.2674994

[10] Pedro Henrique Fernandes and Carlos Baquero. 2023. Probabilistic Causal Contexts for Scalable CRDTs. In *Proceedings of the 10th Workshop on Principles and Practice of Consistency for Distributed Data, PaPoC 2023, Rome, Italy, 8 May 2023*, Elisa Gonzalez Boix and Pierre Sutra (Eds.). ACM, 1–8. https://doi.org/10.1145/3578358.3591331

[11] Andrii Gakhov. 2019. *Probabilistic data structures and algorithms for big data applications.* BoD–Books on Demand.

[12] Google. 2016. Guava Explained. https://github.com/google/guava/wiki. Accessed: May 31, 2024.

[13] Thomas Mueller Graf and Daniel Lemire. 2019. Xor Filters: Faster and Smaller Than Bloom and Cuckoo Filters. *CoRR* abs/1912.08258 (2019). arXiv:1912.08258 http://arxiv.org/abs/1912.08258

[14] Adam Kirsch and Michael Mitzenmacher. 2006. Less Hashing, Same Performance: Building a Better Bloom Filter. In *ESA (Lecture Notes in Computer Science, Vol. 4168)*. Springer, 456–467.

[15] Martin Kleppmann and Alastair R. Beresford. 2017. A Conflict-Free Replicated JSON Datatype. *IEEE Trans. Parallel Distributed Syst.* 28, 10 (2017), 2733–2746. https://doi.org/10.1109/TPDS.2017.2697382

[16] Lightbend. 2011. Akka Documentation, Replicated data types. https://doc.akka.io/libraries/akka-core/current/typed/distributed-data.html#replicated-data-types. Accessed: January 19, 2025.

[17] Mahsa Najafzadeh, Marc Shapiro, Valter Balegas, and Nuno M. Preguiça. 2013. Improving the Scalability of Geo-replication with Reservations. In *IEEE/ACM 6th International Conference on Utility and Cloud Computing, UCC 2013, Dresden, Germany, December 9-12, 2013*. IEEE Computer Society, 441–445. https://doi.org/10.1109/UCC.2013.87

[18] Rasmus Pagh and Flemming Friche Rodler. 2001. Cuckoo Hashing. In *Algorithms - ESA 2001, 9th Annual European Symposium, Aarhus, Denmark, August 28-31, 2001, Proceedings (Lecture Notes in Computer Science, Vol. 2161)*, Friedhelm Meyer auf der Heide (Ed.). Springer, 121–133. https://doi.org/10.1007/3-540-44676-1_10

[19] Kevin De Porre, Carla Ferreira, and Elisa Gonzalez Boix. 2023. VeriFx: Correct Replicated Data Types for the Masses. In *37th European Conference on Object-Oriented Programming, ECOOP 2023, July 17-21, 2023, Seattle, Washington, United States (LIPIcs, Vol. 263)*, Karim Ali and Guido Salvaneschi (Eds.). Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 9:1–9:45. https://doi.org/10.4230/LIPICS.ECOOP.2023.9

[20] Nuno Preguiça, Joan Manuel Marques, Marc Shapiro, and Mihai Letia. 2009. A Commutative Replicated Data Type for Cooperative Editing. In *29th IEEE International Conference on Distributed Computing Systems (ICDCS 2009)*. IEEE, 395–403. https://doi.org/10.1109/ICDCS.2009.20

[21] Nuno M. Preguiça. 2018. Conflict-free Replicated Data Types: An Overview. *CoRR* abs/1806.10254 (2018). arXiv:1806.10254

[22] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. 2011. *A comprehensive study of convergent and commutative replicated data types.* Ph. D. Dissertation. Inria–Centre Paris-Rocquencourt; INRIA.

[23] Marc Shapiro, Nuno M. Preguiça, Carlos Baquero, and Marek Zawirski. 2011. Conflict-Free Replicated Data Types. In *Stabilization, Safety, and Security of Distributed Systems SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6976)*, Xavier Défago, Franck Petit, and Vincent Villain (Eds.). Springer, 386–400. https://doi.org/10.1007/978-3-642-24550-3_29

[24] Haoyu Song, Sarang Dharmapurikar, Jonathan S. Turner, and John W. Lockwood. 2005. Fast hash table lookup using extended bloom filter: an aid to network processing. In *Proceedings of the ACM SIGCOMM 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, Philadelphia, Pennsylvania, USA, August 22-26, 2005*, Roch Guérin, Ramesh Govindan, and Greg Minshall (Eds.). ACM, 181–192. https://doi.org/10.1145/1080091.1080114

[25] Junbo Xiong Trovato. 2024. A Java / Scala library for conflict-free replicated probabilistic filters. https://github.com/C6H5-NO2/probfilter. Accessed: January 19, 2025.

[26] Stéphane Weiss, Pascal Urso, and Pascal Molli. 2009. Logoot: A Scalable Optimistic Replication Algorithm for Collaborative Editing on P2P Networks. In *29th IEEE International Conference on Distributed Computing Systems (ICDCS 2009), 22-26 June 2009, Montreal, Québec, Canada*. IEEE Computer Society, 404–412. https://doi.org/10.1109/ICDCS.2009.75

[27] Junbo Xiong Trovato. 2024. *Conflict-Free Replicated Probabilistic Filter.* Master's Thesis. Delft University of Technology.

[28] Minlan Yu, Alex Fabrikant, and Jennifer Rexford. 2009. BUFFALO: bloom filter forwarding architecture for large organizations. In *Proceedings of the 2009 ACM Conference on Emerging Networking Experiments and Technology, CoNEXT 2009, Rome, Italy, December 1-4, 2009*, Jörg Liebeherr, Giorgio Ventre, Ernst W. Biersack, and Srinivasan Keshav (Eds.). ACM, 313–324. https://doi.org/10.1145/1658939.1658975

[29] Yuqi Zhang, Lingzhi Ouyang, Yu Huang, and Xiaoxing Ma. 2023. Conflict-free Replicated Priority Queue: Design, Verification and Evaluation. In *Proceedings of the 14th Asia-Pacific Symposium on Internetware, Internetware 2023, Hangzhou, China, August 4-6, 2023*, Hong Mei, Jian Lv, Zhi Jin, Xuandong Li, Xiaohu Yang, and Xin Xia (Eds.). ACM, 302–312. https://doi.org/10.1145/3609437.3609452