Conflict-Free Replicated Probabilistic Filter

Junbo Xiong

Conflict-Free Replicated Probabilistic Filter

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Junbo Xiong

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
ewi.tudelft.nl

Conflict-Free Replicated Probabilistic Filter

Author: Junbo Xiong
Student id: 5689937

Abstract

Conflict-free replicated data types (CRDTs) offer high-availability low-latency updates to data without the need for central coordination. Despite the current vast collection of CRDTs, few works have been done on maintaining probabilistic membership information using CRDTs. In this thesis, we fill the gap by proposing conflict-free replicated probabilistic filters (CRPFs). The CRPFs provide probabilistic guarantees similar to that of the Bloom filter and the cuckoo filter, and allow for concurrent updates implementing grow-only and observed-remove semantics. We check their correctness by both paper proof and software verification. Our experiments demonstrate the performance of CRPFs in real-world settings regarding the empirical false positive rate and memory consumption.

Thesis Committee:

Chair:	Prof. Dr. Arie van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. Burcu Özkan, Faculty EEMCS, TU Delft
Committee Member:	Dr. Jérémie Decouchant, Faculty EEMCS, TU Delft
	Ege Berkay Gülcan, Faculty EEMCS, TU Delft

Preface

I'd like to thank Burcu Kulahcioglu Ozkan and Ege Berkay Gulcan for their invaluable guidance throughout the thesis project. I am also grateful to my family and friends for their constant help and support.

Junbo Xiong
Delft, the Netherlands
September 1, 2024

Contents

List of Figures

Chapter 1

Introduction

Membership queries answer the existence of a particular element in a dataset. Many practical problems can be seen as membership queries, such as identifying whether a URL is marked malicious and checking if a username has already been taken. However, in the setting of distributed systems, datasets are usually massive and partitioned. Lookup operations on them are generally slow due to costly disk IOs and network requests. Therefore, precise membership queries are sometimes undesirable.

An alternative approach is introducing approximate membership queries (AMQs) [19]. Here, *approximate* indicates that such queries return `true` for non-existing elements with a certain probability, the false positive rate (FPR). Meanwhile, false negatives are not allowed. Backed by probabilistic filters like the Bloom filter [6], AMQs are typically efficient in both space and time since the filters store compacted information about original datasets into memory and perform queries thereof. Unnecessary lookup operations can be filtered out by conducting approximate queries first and following up with precise ones only when the former responds with `true`. In some use cases, false positive results themselves are accepted, and AMQs can replace the original queries directly.

To support AMQs on distributed data, previous works suggest merging Bloom filters [42, 24]. However, these works regard filters as static representations of local data rather than dynamic data structures that can be actively updated in a distributed manner. For instance, they fail to address the resolution of concurrent updates to the original data. A potential solution is to handle those updates locally before broadcasting them to other replicas. According to the CAP theorem [9, 20], the lower demand on consistency offers higher availability as well as partition tolerance.

Conflict-free replicated data types (CRDTs) [38] are special data structures that allow concurrent updates across distributed replicas without requiring any central coordination. CRDTs provide eventual consistency [39], which guarantees that updates, even if conflicting, can be merged so that all replicas will eventually converge on equivalent states once all updates are delivered to all replicas. In spite of the extensive array of existing CRDTs, there are currently no CRDT versions of probabilistic filters yet.

In this thesis, we try to explore probabilistic filters within the CRDT framework. We propose novel conflict-free replicated probabilistic filters (CRPFs) for solving AMQ problems in distributed systems. We aim to maintain a distributed probabilistic guarantee where

membership queries can yield a bounded probability of false positives but never false negatives. The CRPFs are capable of handling concurrent updates of elements. We focus on two popular types of filters, the Bloom filter and the cuckoo filter, and two concurrency semantics, grow-only and observed-remove. We believe that these conflict-free replicated probabilistic filters will enrich the current CRDT landscape.

1.1 Research Questions

The research questions to be explored are:

- **RQ 1**: What operations does CRPF support?
  - **RQ 1.1**: What operations make CRPF a probabilistic filter?
  - **RQ 1.2**: What operations make CRPF a CRDT?
  - **RQ 1.3**: How is the design verified?
- **RQ 2**: What are the false positive rate and false negative rate of CRPF?
- **RQ 3**: How does CRPF perform empirically?
  - **RQ 3.1**: How is the empirical false positive rate?
  - **RQ 3.2**: How is the memory consumption?

1.2 Contributions

The contributions of this thesis are listed below.

1. Specifications of 6 conflict-free replicated probabilistic filters.
2. An analysis of false positive rate, both theoretical and empirical.
3. An open-source implementation of CRPFs and the verification process.

1.3 Thesis Outline

The thesis is structured as follows. Chapter 2 presents some background information on CRDTs and probabilistic filters. In Chapter 3, we propose the specifications of our CRPFs. We explain how we verify their correctness in Chapter 4 and how we implement them in Chapter 5. Chapter 6 shows the experiment results and our analyses. Chapter 7 covers some relevant literature.

Chapter 2

Background

This chapter explains the two central concepts in this thesis: conflict-free replicated data types and probabilistic filters.

2.1 Conflict-Free Replicated Data Types

Conflict-free Replicated Data Types (CRDTs) [38, 39] are a group of data structures that allow for concurrent updates in distributed systems without central coordination. Potential conflicts introduced by the updates are resolved automatically so that strong eventual consistency is guaranteed. With this consistency model, all replicas reach equivalent states if receiving the same updates [39]. Current CRDT implementations encompass a wide range of data types, including registers [38], counters [38], sets [38, 4], graphs [38], trees [32], ordered sequences [36], maps [11], priority queues [44], HyperLogLog [35], and JSON documents [27]. In the following subsections, we describe approaches to replicating CRDTs and strategies to make them conflict-free.

2.1.1 Replication Model

[39] introduces two approaches to modelling CRDT: state-based and operation-based. The state-based approach formalises CRDT as join semilattice [5]. The states of a state-based CRDT take their values from the lattice and are periodically propagated to other replicas. The recipients then *merge* the received states with their own ones, which is defined to give the least upper bound of the states. The states are *updated* by mutator operations. Every mutator operation executes locally and transforms the input state into a greater one. This way, the states evolve monotonically and converge towards the upper bound. State-based CRDTs do not require a highly reliable network, thanks to the properties of the least upper bound [39]. The states can be delivered more than once, in any order, or even indirectly through other states.

The operation-based model views CRDT as the effect of a sequence of operations. Op-based CRDTs broadcast operations through a reliable channel that guarantees exactly-once casual delivery, different from state-based ones that send the entire state. The operations are then applied to local states of the replicas to produce new states as *effects*. [39] shows that

if all concurrent operations commute, operation-based CRDTs will eventually converge. Despite their differences, the two models above are equivalent in semantics. One model can be emulated using the other.

2.1.2 Conflict Resolution Strategy

Several strategies have been proposed to resolve conflicting operations in CRDTs like concurrent *add* and *remove* of the same element. These strategies are implemented as various *merge* functions in state-based CRDTs or as *effect* phases in operation-based CRDTs.

A trivial strategy is grow-only, which allows only a single type of monotonic mutation and therefore avoids conflicts from the beginning. For example, a grow-only set [38] supports only element addition, and a grow-only counter [38] can only be increased. Since the state grows monotonically, concurrent operations are directly aggregated by taking the maximum, and no conflict will occur.

To support both adding and removing elements from a CRDT collection, [38] presents a strategy called observed-remove. Like its name suggests, when an *add* conflicts with a *remove*, *add* will take effect unless *remove* observes it. The *observe* here means that the *remove* happens-after the *add* in the causal history. This strategy is also known as add-wins since concurrent *adds* win against *removes*. Similarly, there is also the remove-wins strategy [4], which lets concurrent *removes* take precedence over *adds*.

Another choice of arbitrating between conflicting operations is by comparing their timestamps [29]. In the last-writer-wins strategy [38], operations are associated with unique timestamps to form a total order so that the last operation according to this order determines the final state.

Specific data types can implement their dedicated ways to resolve conflicts. For example, multi-value register [38] keeps all the results of concurrent updates, which can then be reduced into one single value through a subsequent assignment. [27] designs a CRDT for JSON documents using a strategy aiming at preserving user input, even for concurrent modifications in different levels of the document tree.

2.2 Probabilistic Filters

A probabilistic filter is an approximate but compact representation of a set. Compared to ordinary sets that answer membership queries accurately, probabilistic filters have long been the solution to approximate membership problems [16, 19]. In this section, we describe two popular types of probabilistic filters and a technique that allows for unbounded capacity.

2.2.1 Bloom Filter

The Bloom filter [6] uses an array of m bits to represent the elements of a dataset. Initially, all bits are 0. When an element e is added, it is first hashed by k independent hash functions

$h_1, h_2, \cdots, h_k$ to calculate indexes in the array:

$$\begin{aligned} i_1 &= h_1(e), \\ i_2 &= h_2(e), \\ &\vdots \\ i_k &= h_k(e). \end{aligned} \tag{2.1}$$

The bits in those indexes are then set to 1. Querying the membership status of a certain element follows a similar procedure by examining whether all the corresponding bits are 1. If so, the Bloom filter reports that the element belongs to the dataset. Otherwise, the element is not part of the dataset.

Since all relevant bits are set and removal is not supported, there is no false negative in the standard Bloom filter. False positive, on the other hand, is possible because a bit can be set by any other elements. After adding n elements, the false positive rate is [6]:

$$\varepsilon = \left[1 - (1 - \frac{1}{m})^{kn}\right]^k \approx (1 - e^{-kn/m})^k. \tag{2.2}$$

Given this formula, one can construct a Bloom filter with an expected FPR ε and capacity n. The optimal number of bits m_{opt} and number of hash functions k_{opt} are the values that minimise FPR [19]:

$$m_{\text{opt}} = \frac{-n \ln \varepsilon}{(\ln 2)^2}, \tag{2.3}$$

$$k_{\text{opt}} = \frac{-\ln \varepsilon}{\ln 2}. \tag{2.4}$$

2.2.2 Cuckoo Filter

Unlike the Bloom filter that uses indexes in a bit array to record elements, the cuckoo filter [17] stores elements as fingerprints. It comprises an array of m buckets, and each bucket contains c slots for the fingerprints. The fingerprint f of an element is a short bit sequence of length l generated by a fingerprint function:

$$f = h_F(e). \tag{2.5}$$

The cuckoo filter adopts a partial-key cuckoo hashing algorithm [17] for locating the fingerprint of an element. It maps an element to two possible candidate buckets. One candidate index i_1 is calculated by an index hash function:

$$i_1 = h_I(e); \tag{2.6}$$

the other index i_2 results from:

$$i_2 = altIndexOf(i_1, f). \tag{2.7}$$

The function *altIndexOf* is an involution given fingerprint f satisfying:

$$altIndexOf(altIndexOf(i, f), f) \equiv i, \tag{2.8}$$

so that i_1 can be derived from i_2 and vice versa. The authors of the original cuckoo filter paper propose an involution based on bit-wise XOR [17]:

$$altIndexOf(i, f) = i \oplus h_I(f), \tag{2.9}$$

yet other implementations are also possible, for instance, one utilising (modular) reflection [25].

Based on such an algorithm, an element can be added by storing its fingerprint f in either of its candidate buckets that has available slots. If all of the slots are occupied, a random fingerprint will be evicted from those slots and f takes its place. The victim fingerprint is then displaced to its alternative bucket, where the above operations repeat until a slot is found or a maximal number of iterations is reached. Since any element will end up being a fingerprint in its candidate buckets, element removal and membership query are straightforward. The former boils down to removing one instance of the fingerprint (if any), and the latter is equivalent to checking the existence of the fingerprint.

Due to hash collisions in fingerprint and bucket indexes, membership queries with the cuckoo filter do have false positives. The upper bound of FPR is given by [17]:

$$\varepsilon = 1 - (1 - \frac{1}{2^l})^{2c} \approx \frac{2c}{2^l}. \tag{2.10}$$

Moreover, false negatives are not possible in the cuckoo filter if all removal operations are "safe". That is, removal should only be called with previously added elements [17].

2.2.3 Scalability

Both the Bloom filter and the cuckoo filter have a fixed limit upon the number of elements they can hold. Such capacity is a parameter that needs to be specified at initialisation. If more elements are added to the Bloom filter, its FPR will deteriorate according to Equation 2.2. The cuckoo filter will run out of its slots in such cases. To allow probabilistic filters to handle additions more than the original estimation, [1] proposes scalable Bloom filter. It uses a series of normal Bloom filters as sub-filters, and membership queries are delegated to these sub-filters. Despite that the authors only apply the technique to the Bloom filter, we argue that the principle is independent of specific filters and can be extended to others.

The scalable filter offers a bounded compounded FPR. Denote the FPR of the first sub-filter as ε_0. The FPR of the i-th sub-filter equals that of its predecessor times a tightening ratio q:

$$\varepsilon_i = \varepsilon_0 q^i. \tag{2.11}$$

The ratio q is a value between 0 and 1. A typical choice is $q = 0.5$ [1]. For a scalable filter containing w such sub-filters, the compounded FPR is thus the sum of a geometric

series [1]:

$$\varepsilon = 1 - \prod_{i=0}^{w-1} (1 - \varepsilon_0 q^i) \leq \frac{\varepsilon_0}{1-q} = 2\varepsilon_0 \,. \tag{2.12}$$

Therefore, by building the first sub-filter with an FPR half of the expected compounded value ε, we can construct a scalable filter whose final FPR is bounded by ε.

Using Equation 2.3 and Equation 2.4, a Bloom filter can be created given the expected FPR as a sub-filter. To halve the FPR of a cuckoo filter, one additional bit is allocated for the fingerprint.

Chapter 3

Specification

This chapter describes the specifications of our conflict-free replicated probabilistic filters (CRPFs). We first discuss the system model in Section 3.1 and then introduce the new CRDTs in the following sections. We aim to show in detail what operations the CRPFs need to support and analyse their probabilistic characteristics.

3.1 System Model

We adopt a system model similar to that in standard CRDT settings [38]. The distributed system is formed from a number of replicas connected by an asynchronous network. Each replica hosts one instance of the CRDT and is assigned a unique ID. It may crash but will eventually recover without corrupting its state. The network may also crash, lead to partition, but will eventually recover. We do not impose special restrictions on the network; messages may be delivered more than once and/or out of order. The only requirement is that they should eventually arrive at their recipients. Malicious behaviour and Byzantine faults are beyond the scope of the thesis. Note that there is no central server in the network, and all replicas operate without any global coordination among them.

In this system, we expect to maintain a distributed probabilistic guarantee of membership status. The result of a membership query may be false positive but should never be false negative.

We use the state-based approach [38] for specifying our CRPFs. Specification 1 outlines a common interface of CRPF. We use the keyword *payload* to declare the state at each replica and supply it with the initial value specified by *initial*. A *query* is a purely local procedure that outputs a result without modifying the state. CRPF only has one *query* named *contains* that checks whether an element is likely to be a member of the filter. An *update* first changes the state at the source replica and then broadcasts the updated state asynchronously. Note that, for clarity, we assume all types to be immutable in the specifications, though they can be implemented otherwise. Every "mutation" returns a new instance rather than directly mutating the old one in place. Hence, the return type of every *update* is the payload type. There are two *updates* in CRPF, one for adding an element and the other for removing if supported.

Specification 1: Common interface representing state-based CRPF

```
1 payload payload_name: payload_type
2 initial initial value of payload
3 query contains(e: element_type): bool
4 update add(e: element_type): payload_type
5 update remove(e: element_type): payload_type // Not all filters need to
      support removal.
6 compare (other: payload_type): bool
7 merge (other: payload_type): payload_type
```

The function *compare* defines a partial order of the states. It returns `true` if the state of the current replica is partially less than or equal to `other` and `false` otherwise. Meanwhile, *merge* yields the least upper bound of the current state and `other` given the partial order. In practice, *merge* is called when a message (i.e. state) is received, while *compare* is never actively invoked.

3.2 Grow-Only Bloom Filter

We propose our conflict-free replicated Bloom filter design: the grow-only Bloom filter, abbreviated as GBF. Since it supports only adding elements, we base our design on the grow-only set (GSet) [38]. Specification 2 represents a state-based grow-only Bloom filter consists of an m-bit bit array and k hash functions $\{h_i\}_{i=1}^{k}$. We model the bit array as a conceptual set of active indexes. Initially, all bits are set to 0, hence the empty set.

3.2.1 Supported Operations

Query and update operations in the GBF are the same as the corresponding ones in the conventional Bloom filter. The addition of an element e is equivalent to a set union of the current state and the corresponding set of hash indexes of e calculated using $\{h_i(e)\}_{i=1}^{k}$. If the set is the subset of the current state, e is considered to be contained by the filter.

3.2.2 State Convergence

Likewise, the set union of the states of two GBFs yields their merged state, given that the partial $\leq$ relation is defined as a set inclusion relation. Figure 3.1 illustrates a simple case of adding elements and merging states of two GBFs each having two hash functions. In this example, the merged state contains all active indexes introduced by e_1 and e_2. As a result, membership queries about both elements yield `true`.

3.2.3 Probabilistic Characteristics

The false positive rate of the standard Bloom filter, shown in Equation 2.2, depends on the number of bits m, the number of hash functions k, and the number of inserted elements.

Specification 2: State-based grow-only Bloom filter

```
1  payload S: set⟨index⟩
2  initial S ← ∅
3  query contains(e):
4  |  return {h_i(e)}_{i=1}^k ⊆ S
5  end
6  update add(e):
7  |  return {h_i(e)}_{i=1}^k ∪ S
8  end
9  compare (S′):
10 |  return S ⊆ S′
11 end
12 merge (S′):
13 |  return S ∪ S′
14 end
```

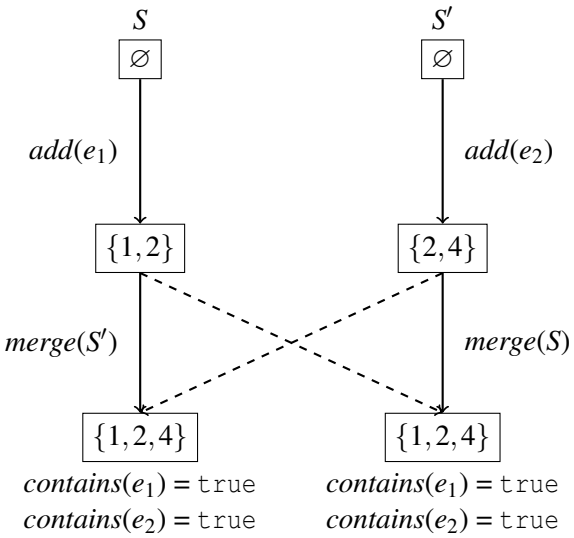


Figure 3.1: An example of merging two GBFs. In this example, e_1 is hashed to $\{1,2\}$ and e_2 to $\{2,4\}$. The merged states contain both elements.

As for our grow-only Bloom filter, the difference in the formula for FPR only lies in the last parameter since m and k are identical for all replicas and the hashing procedure is not modified. In GBF, the total number of insertions that happen-before the query is used. We denote the sum as $\sum n$. The false positive rate ε is thus bounded by:

$$\varepsilon = \left[1 - (1 - \frac{1}{m})^{k\sum n}\right]^k \approx (1 - e^{-k(\sum n)/m})^k. \tag{3.1}$$

The false negative rate is zero, identical to the standard Bloom filter. GBF offers a probabilistic guarantee similar to its conventional counterpart and is therefore suitable for direct substitution.

3.3 Grow-Only Cuckoo Filter

As an alternative to the Bloom filter, we put forward a cuckoo filter implementing the grow-only strategy, the grow-only cuckoo filter (GCF), by augmenting the grow-only set [38]. We elaborate in the following paragraphs Specification 3, the specification of a state-based grow-only cuckoo filter having m buckets, each *expected* to store no more than c fingerprints.

Similar to the conventional cuckoo filter, our grow-only cuckoo filter holds a cuckoo hash table that is represented 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 precise slot is irrelevant since they are all homogeneous. The algorithms for calculating the fingerprint and bucket indexes are all identical to those of the standard cuckoo filter, which are defined in Equation 2.5, Equation 2.6, and Equation 2.7.

A major difference is that the capacity c of a bucket is a hard limit in the standard cuckoo filter, whereas it becomes a soft one in the grow-only cuckoo filter. This is because a bucket may contain more entries than expected in GCF. A bucket containing exactly c entries is considered *full*. It is regarded as *overflowing* if more entries are inserted. We delay the reason for allowing overflowing until Subsection 3.3.2. Regarding the implementation, an array is used to store the regular entries, and a map is added to accommodate the overflowing parts.

3.3.1 Supported Operations

Both of its candidate buckets are inspected to query the existence of an element e. If either one contains its fingerprint, a positive result is returned. This step is crucial before inserting fingerprints into the table to avoid duplicated entries. The element will only be added if it has not been contained yet.

Due to the difference in bucket capacity constraint, GCF adopts an alternative approach to adding elements. Algorithm 1 defines *cuckooInsert*, a novel function for inserting entries into a cuckoo hash table that takes into account overflowing buckets. The initial steps do not vary greatly, in which both candidate buckets are examined for available slots. The one that has empty slots is preferred. Otherwise, a random bucket is chosen. Then, it tries to insert the entry into the chosen bucket.

The following steps in *cuckooInsertImpl* form a repeating procedure that moves entries around to accommodate the newly inserted one. If the current bucket is not full, the entry is inserted into an available slot there, and the procedure completes successfully. If the bucket is just full, the entry replaces a random one in the bucket, and then the victim entry continues with *cuckooInsertImpl* in its alternative bucket. If the bucket is overflowing, the algorithm first tries "smoothing out" the excess entries by displacing them. Then, the remaining quotas for iterations are spent on inserting the original entry. *cuckooInsertImpl* aborts if it runs out of quota, which ensures that the process terminates within a reasonable amount of time. In this way, we make sure that no overflowing bucket is introduced due to local *adds* and prevent accumulating even more entries in existing ones. Furthermore, the algorithm redistributes the entries on the fly and rebalances the cuckoo hash table, which is expected to improve the load factor.

Specification 3: State-based grow-only cuckoo filter

payload S: set$\langle$index $\times$ fingerprint$\rangle$
initial $S \leftarrow \varnothing$
query *contains(e)*:
    $f \leftarrow h_F(e)$
    $i_1 \leftarrow h_I(e)$
    $i_2 \leftarrow altIndexOf(i_1, f)$
    return $(i_1, f) \in S \vee (i_2, f) \in S$
end
update *add(e)*:
    if *contains(e)* **then**
        return S
    end
    $f \leftarrow h_F(e)$
    $i_1 \leftarrow h_I(e)$
    return $cuckooInsert(S, i_1, f)$ `// No auxiliary data is needed in GCF.`
end
compare *(S′)*:
    $D \leftarrow \{(altIndexOf(i,f), f) \mid (i,f) \in S\}$ `// The dual set`
    $D' \leftarrow \{(altIndexOf(i,f), f) \mid (i,f) \in S'\}$
    return $(S \cup D) \subseteq (S' \cup D')$
end
merge *(S′)*:
    $D \leftarrow \{(altIndexOf(i,f), f) \mid (i,f) \in S\}$
    return $S \cup (S' \setminus D)$
end

Algorithm 1: Static function that insert into the bucket at index i_1 of state S an entry made up of fingerprint f and optional auxiliary data x

```
1  def cuckooInsert(S, i1, (f, x)):
2      i2 ← altIndexOf(i1, f)
3      B1 ← {(i1, f′, x′) ∈ S}
4      B2 ← {(i2, f′, x′) ∈ S}
5      if |B1| < c ∧ |B2| ≥ c then
6          i ← i1
7      else if |B2| < c ∧ |B1| ≥ c then
8          i ← i2
9      else
10         i ← randomChoose({i1, i2})
11     end
12     a ← quota for iteration
13     (S′, _) ← cuckooInsertImpl(S, i, (f, x), a)
14     return S′
15 end
16 def cuckooInsertImpl(S, i, (f, x), a):
17     if a < 0 then
18         throw "maximum number of iterations is reached"
19     end
20     B ← {(f′, x′) | (i, f′, x′) ∈ S}
21     if |B| < c then
22         // Insert into an available slot directly
23         return S ∪ {(i, f, x)}, a
24     else if |B| = c then
25         // Displace a random entry and shift it to its alt bucket
26         (f′, x′) ← randomChoose(B)
27         S′ ← S \ {(i, f′, x′)} ∪ {(i, f, x)}
28         i′ ← altIndexOf(i, f′)
29         return cuckooInsertImpl(S′, i′, (f′, x′), a − 1)
30     else
31         // Evict excess entries repeatedly and try rebalancing the
               table
32         (f′, x′) ← randomChoose(B)
33         S′ ← S \ {(i, f′, x′)}
34         i′ ← altIndexOf(i, f′)
35         (S″, a″) ← cuckooInsertImpl(S′, i′, (f′, x′), a − 1)
36         return cuckooInsertImpl(S″, i, (f, x), a″)
37     end
38 end
```

3.3.2 State Convergence

Since the same fingerprint appearing in either candidate bucket indicates the existence of the same element (ignoring hash collisions), a cuckoo hash table S corresponds to a larger conceptual set of entries. This conceptual universe 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:

$$D(S) = \{(altIndexOf(i, f), f) \mid (i, f) \in S\}. \tag{3.2}$$

Then, we get:

$$U(S) = S \cup D(S). \tag{3.3}$$

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

$$S \leq S' \Leftrightarrow U(S) \subseteq U(S'). \tag{3.4}$$

Intuitively, the *merge* operation should be a set union:

$$S \sqcup S' = S \cup S'. \tag{3.5}$$

Despite being theoretically correct, the naïve solution leads to the duplication of entries corresponding to the same element. Figure 3.2a demonstrates an example case. An observation is that one cuckoo hash table only needs to retain entries that are not covered by its dual table because those entries are already included implicitly. Therefore, we can explicitly delete entries in the dual table whilst still reaching an equivalent state. The *merge* function is thus adapted to be:

$$S \sqcup S' = S \cup (S' \setminus D(S)). \tag{3.6}$$

A step-by-step illustration of the function is depicted in Figure 3.2b.

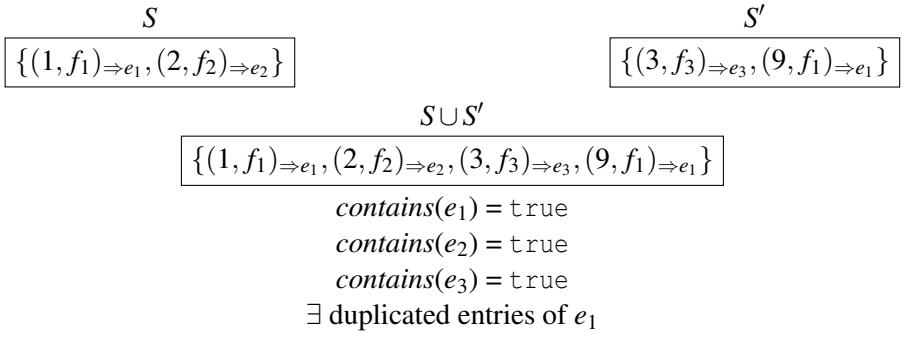
Although local *adds* do not cause buckets to overflow, it is possible that several non-overflowing buckets are *merged* into an overflowing one. Since checking the bucket size beforehand is not possible without global coordination and the definition of CRDT requires *merge* to be always enabled [38], we have to allow overflowing buckets in *merge*. In spite of this, we leverage the special *cuckooInsert* algorithm to mitigate the issue.

3.3.3 Probabilistic Characteristics

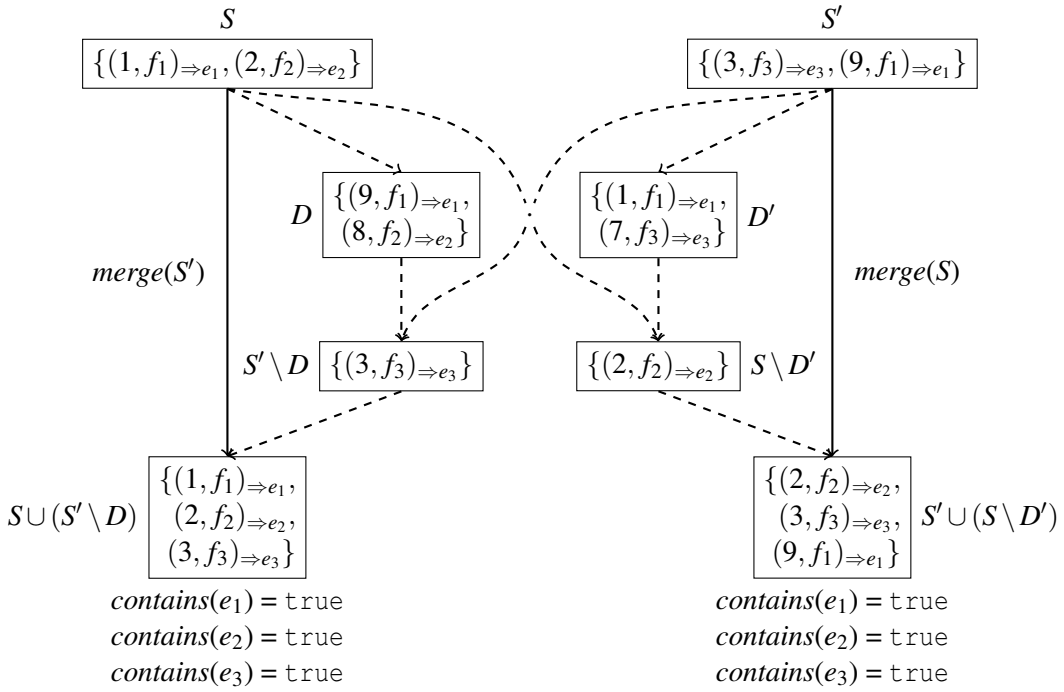
As mentioned above, there are chances when the load factor α of some bucket exceeds 100% in GCF. For this reason, we incorporate the average load factor of the table $\bar{\alpha}$ into the estimation of FPR. In a conventional cuckoo filter, at most $2c$ entries are compared with the expected fingerprint, hence the term in Equation 2.10. In GCF, $2c\bar{\alpha}$ slots are accessed per membership query on average. Therefore, the FPR of a GCF using l-bit fingerprint is approximately:

$$\varepsilon = 1 - (1 - \frac{1}{2^l})^{2c\bar{\alpha}} \approx \frac{2c\bar{\alpha}}{2^l}. \tag{3.7}$$

There are no false negatives in GCF.



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



(b) The detailed procedure of *merge* using Equation 3.6. There is no duplicated entry.

Figure 3.2: An example of merging two GCFs using different implementations. 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. However, only the method in Figure 3.2b succeeds in eliminating duplication.

3.4 Observed-Remove Cuckoo Filter

The previous CRPFs only allow insertions. In this section, we propose a conflict-free replicated cuckoo filter that also allows the removal of elements. We adopt the observed-remove strategy [38] in order to specify the precedence of concurrent *adds* and *removes*. Thus, we extend the optimised observed-remove set (ORSet) [4] to design our observed-remove cuckoo filter, ORCF for short.

Specification 4 shows the specification of the state-based observed-remove cuckoo filter. Its payload is composed of a cuckoo hash table S and a causal history H. An entry in S now contains two more fields in addition to the fingerprint f, including a replica id r and a logical timestamp t. These two fields form a version tag that records the source replica and the time the entry is created. Since the replica ID is unique and the timestamp grows monotonically, the tags act as globally unique yet partially ordered identifiers that help reconstruct the causal order of operations. All tags that the replica has observed are stored in the causal history H. In the specification, H is modelled as a set of tags, but in practice it can be encoded as a version vector [34, 31], storing only the maximum observed timestamp per replica.

3.4.1 Supported Operations

The ORCF supports membership query, element addition, and element removal. The membership query is slightly modified compared to that of the classical cuckoo filter. It checks whether there is an entry whose fingerprint component is equal to the fingerprint f of an element e in either of the element's candidate buckets i_1 and i_2.

During element addition, the initiating replica's ID r and the next available timestamp t are associated with the fingerprint f to build an entry. The entry is then inserted into the cuckoo hash table using the same *cuckooInsert* function in Algorithm 1. Meanwhile, the new version tag (r,t) is added to the causal history. Note that, in contrast to the GCF, we do not check whether an element is already contained before adding it to an ORCF. We allow the cuckoo hash table to retain multiple copies of equivalent entries of an element, as required by deletable cuckoo filters [17].

Removal of an element starts by finding all entries corresponding to the element. Then, a random one among those is deleted from the cuckoo hash table. This is a routine procedure for ordinary cuckoo filters. However, it has a downside under distributed settings, where it may impact upon the probabilistic characteristics of the ORCF. We will cover this topic in Subsection 3.4.3. Considering that *remove* does not create new entries or version tags, the causal history is left untouched.

3.4.2 State Convergence

We first set up the partial order of the states. By definition, all *updates*, including *add* and *remove*, should yield a greater state than its input. We notice that the causal history H is only expanded by *add* and is not altered by *remove*. Thus, the partial order related to *add* is:

$$H \subseteq H'. \tag{3.8}$$

Specification 4: State-based observed-remove cuckoo filter

payload S: set$\langle$index $\times$ (fingerprint $\times$ replica_id $\times$ timestamp)$\rangle$, H: set$\langle$replica_id $\times$ timestamp$\rangle$ `// The set H is also known as casual history`
initial $S \leftarrow \varnothing$, $H \leftarrow \varnothing$
query *contains(e)*:
    $f \leftarrow h_F(e)$
    $i_1 \leftarrow h_I(e)$
    $i_2 \leftarrow altIndexOf(i_1, f)$
    return $\exists (r,t) : (i_1, f, r, t) \in S \vee (i_2, f, r, t) \in S$
end
update *add(e)*:
    $f \leftarrow h_F(e)$
    $i_1 \leftarrow h_I(e)$
    $r \leftarrow$ replica id
    $t \leftarrow max(\{t \mid (r,t) \in H\} \cup \{0\}) + 1$ `// The next available timestamp`
    $S' \leftarrow cuckooInsert(S,\ i_1,\ (f,r,t))$
    $H' \leftarrow H \cup \{(r,t)\}$
    return S', H'
end
update *remove(e)*:
    $f \leftarrow h_F(e)$
    $i_1 \leftarrow h_I(e)$
    $i_2 \leftarrow altIndexOf(i_1, f)$
    $R \leftarrow \{(i_1, f, r, t) \in S\} \cup \{(i_2, f, r, t) \in S\}$
    $S' \leftarrow S \setminus \{randomChoose(R)\}$
    return S', H
end
compare *((S′, H′))*:
    $T \leftarrow \{(r,t) \in H \mid \nexists (i,f) : (i, f, r, t) \in S\}$ `// The tombstone set`
    $T' \leftarrow \{(r,t) \in H' \mid \nexists (i,f) : (i, f, r, t) \in S'\}$
    return $H \subseteq H' \wedge T \subseteq T'$
end
merge *((S′, H′))*:
    $S_1 \leftarrow S \cap S'$ `// Not updated`
    $S_2 \leftarrow \{(i, f, r, t) \in S \setminus S' \mid (r,t) \notin H'\}$ `// Exclusive to S and not observed by H′`
    $S_3 \leftarrow \{(i, f, r, t) \in S' \setminus S \mid (r,t) \notin H\}$ `// Exclusive to S′ and not observed by H`
    $S_4 \leftarrow \{(i, f, r, t) \in S \mid (altIndexOf(i, f), f, r, t) \in S'\}$ `// Displaced`
    $S'' \leftarrow S_1 \cup S_2 \cup S_3 \cup S_4$
    $H'' \leftarrow H \cup H'$
    return S'', H''
end

That is, a smaller state observes a subset of the tags observed by a greater state.

As for *remove*, we need a set that grows as *remove* is called to establish the partial order. We can derive a conceptual *tombstone set* that records the version tags of deleted entries [4]. The tombstone set T is defined as:

$$T(S,H) = \{(r,t) \in H \mid \nexists (i,f) \colon (i,f,r,t) \in S\}. \tag{3.9}$$

T is merely a conceptual set required only by *compare*. It is not stored as part of the payload, thanks to the causal history H. Suppose a version tag is contained in the causal history, i.e. observed, but is not part of any entry in the cuckoo hash table. In that case, it can be inferred that the original entry this tag was associated with must have been previously observed and then removed. With the help of the tombstone set, the partial order reflecting *remove* is modelled as:

$$T(S,H) \subseteq T(S',H'). \tag{3.10}$$

The equation indicates that a smaller state cannot delete more entries than a greater state.

The integrated formula for *compare* is a logical conjunction of the two above:

$$(S,H) \leq (S',H') \Leftrightarrow H \subseteq H' \ \wedge \ T(S,H) \subseteq T(S',H'). \tag{3.11}$$

The observed-remove strategy requires that the merged state should encompass all active entries expect those that are deleted after being observed. The preserved entries can be categorised into four types based on their origins:

1. Entries not *updated* since the last *merge*, lying in the intersection of the two merging tables.
$$S_1 = S \cap S' \tag{3.12}$$

2. Entries exclusive to the first input ORCF but not observed by the other. Those satisfying this condition are newly added and thus need to be kept. In contrast, exclusive yet observed entries are deleted by the other ORCF and should not be included in the merged state.
$$S_2 = \{(i,f,r,t) \in S \setminus S' \mid (r,t) \notin H'\} \tag{3.13}$$

3. The dual of the above: entries exclusive to the second input ORCF but not yet observed by the first one.
$$S_3 = \{(i,f,r,t) \in S' \setminus S \mid (r,t) \notin H\} \tag{3.14}$$

4. Displaced entries. They can be regarded as a special case of the first type of entries.
$$S_4 = \{(i,f,r,t) \in S \mid (\mathit{altIndexOf}(i,f),f,r,t) \in S'\} \tag{3.15}$$

The final merged cuckoo hash table is the union of the four cases above. The merged causal history is a set union of the two inputs. The *merge* operation of ORCF is therefore:

$$(S,H) \sqcup (S',H') = (S_1 \cup S_2 \cup S_3 \cup S_4,\ H \cup H'). \tag{3.16}$$

An example of *merge* is shown in Figure 3.3.

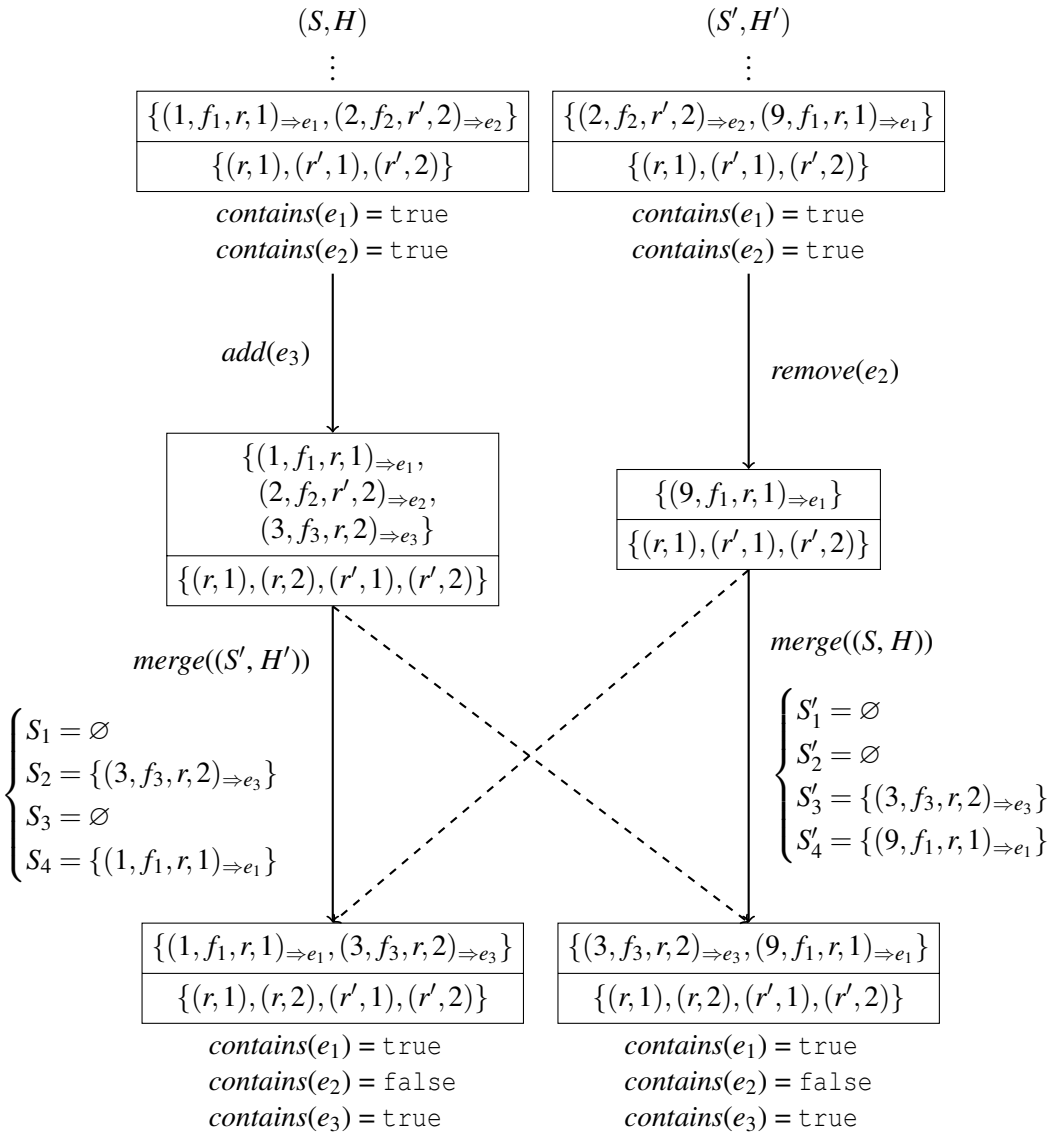


Figure 3.3: An example of merging two ORCFs using the same elements in Figure 3.2. The example starts from a state that contains e_1 and e_2. When *merge* is called, e_1 is merely displaced and is not *updated*, corresponding to case 4. e_3 is newly added, i.e. case 2 (or case 3). Therefore, the two are present in the merged state. However, e_2 is observed and is then removed, hence its absence.

3.4.3 Probabilistic Characteristics

The false positive rate and false negative rate can both be affected by the *remove* operations that happen-before the membership query. We identify two scenarios where concurrent *removes* lead to states that never occur in non-replicated cuckoo filters. Both of them involve removal of elements that are hash collisions. Those elements differ in value but share identical fingerprints and candidate buckets. Figure 3.4 demonstrates the possible outcomes of *removes* after *adds* of colliding elements e_4 and e_5 have been propagated to two ORCFs. We analyse how they interfere with the probabilistic characteristics of the ORCF.

The ordinary cuckoo filter exhibits a zero probability of false negatives if all *removes* are safe [17] from a local viewpoint. However, such behaviour is not guaranteed in the ORCF. Figure 3.4a illustrates a scenario where locally-safe removals result in false negatives in distributed settings. In the figure, both replicas concurrently issue the same operation $remove(e_4)$. From their local perspectives, both replicas view the *remove* as safe because they are removing an element that has been previously added. For this reason, the merged filter is expected to remain well-formed, having a zero false negative rate. But this might not necessarily happen. Due to the issue that e_4 and e_5 are hash collisions, both entries are matched when the set R in Line 22 of Specification 4 is built during the removal of e_4. Each replica then selects a random entry from R to delete. However, it is possible that the two replicas choose different entries to delete. In this case, the merged state does not contain any entry. The membership query about e_5 now gives `false`, which is a false negative since e_5 was never explicitly removed by a $remove(e_5)$.

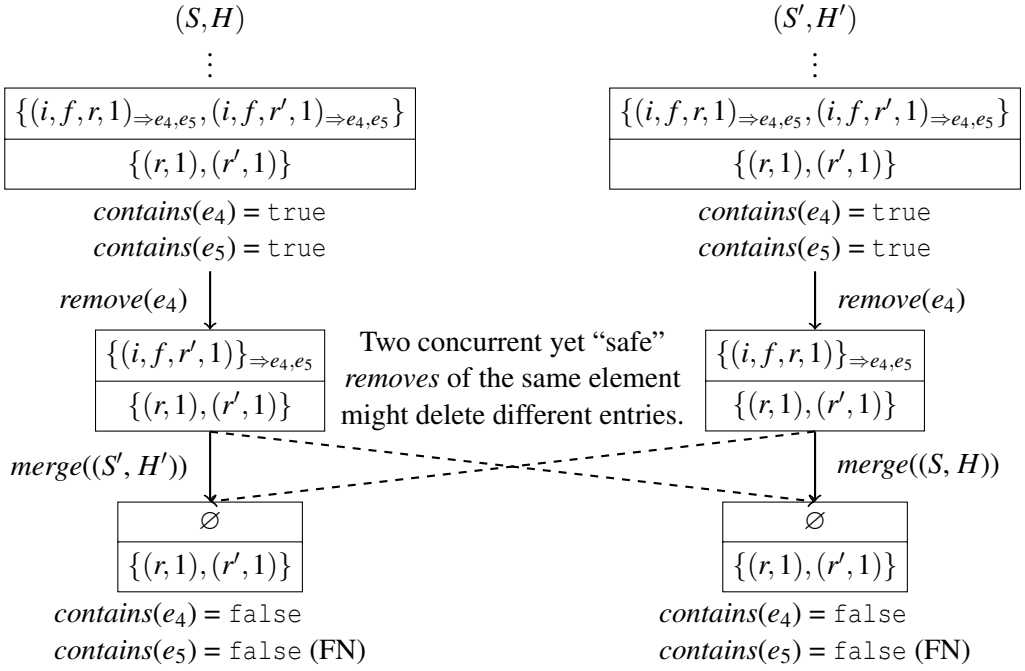
Therefore, we require that all *removes* be *casually-safe* in order to eliminate false negatives in ORCF.

Definition 3.1 (Casually-safe removal)**.** *An operation op removing element e is causally-safe if the number of remove(e) that happen-before or are in parallel with op is less than the number of observed add(e) in the operation history $\mathbb{O}$. It is equivalent to examining whether:*

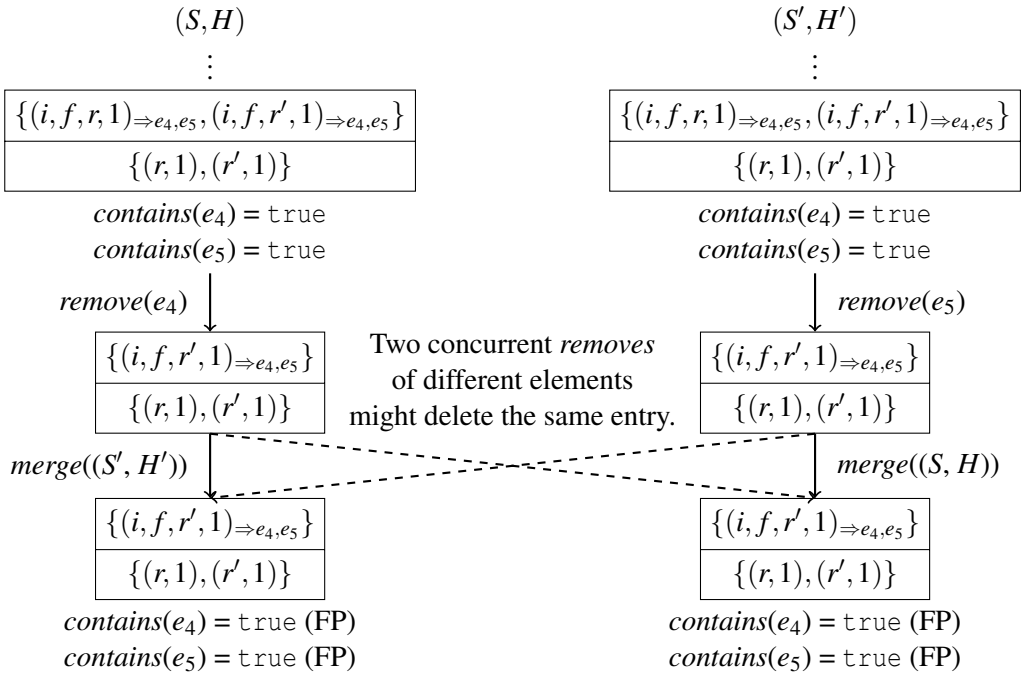
$$\Big|\{remove(e) \in \mathbb{O} \mid remove(e) \prec op \ \vee \ remove(e) \parallel op\}\Big| < \Big|\{add(e) \in \mathbb{O} \mid add(e) \prec op\}\Big|. \tag{3.17}$$

A removal operation *op* satisfying Equation 3.17 will not delete entries brought by other elements, even in the presence of hash collisions. If all *ops* meet the equation, no false negative will occur. Recalling Figure 3.4a, we find that both sides of Equation 3.17 are equal to 1 for the two *removes*. Neither of them is casually-safe, which explains the false negatives in the merged state.

In the other scenario, *remove* contributes to a rise in the false positive rate. Depicted in Figure 3.4b, the two replicas request removal of e_4 and e_5 respectively. As a result of hash collision, they build an identical set R that consists of both entries. Unfortunately, it can happen that they select the same entry to delete, leaving one entry in the merged state. In the example, the entry introduced by e_4 is deleted. Nevertheless, deleting the entry brought by e_5 has the same effect, as the filter cannot differentiate between them. In the merged



(a) Safe *removes* [17] from a local perspective cause false negatives.



(b) Two different *removes* collide, resulting in false positives.

Figure 3.4: Scenarios where *remove* affects the probabilistic characteristics of the ORCF. Both involve hash-colliding elements e_4, e_5 where $e_4 \neq e_5$ but $h_F(e_4) = h_F(e_5) = f$ and $h_I(e_4) = h_I(e_5) = i$. Note that the correctness of the ORCF as a CRDT is intact.

state, membership queries about the two elements both return `true` in spite of two different *removes* because of hash collision. The result is regarded as an increase in the FPR.

The increment of the FPR is reflected in the increment of the load factor. Therefore, we can still abuse $\bar{\alpha}$ to denote the average load factor and approximate the FPR of ORCF as:

$$\varepsilon = 1 - (1 - \frac{1}{2^l})^{2c\bar{\alpha}} \approx \frac{2c\bar{\alpha}}{2^l}. \tag{3.18}$$

We believe that those removals have minimal impact on the FPR. For the problem to occur, three conditions must be met simultaneously:

1. The elements to remove must be hash collisions.
2. The *remove* operations must be parallel.
3. The replicas must independently select the same entry.

In practice, a good hash function typically has a low collision rate. The causal order of operations depends on the actual workload, but in general fewer operations are in parallel as the CRDTs are merged more frequently. The third condition is a birthday problem [37]. Suppose there are d parallel *removes* deleting hash-colliding elements, and there are x entries matched by those elements. The probability that at least two *removes* pick the same entry is:

$$P = 1 - \prod_{k=1}^{d-1}(1 - \frac{k}{x}), \text{ where } d \leq x. \tag{3.19}$$

Since the chances of the first two are small already, we deem that d is also small in most cases. For $d = 2$, P is no larger than 0.5. Consequently, there is little likelihood of all three conditions being satisfied at the same time, and the FPR does not tend to increase much.

3.5 Scalable Filter Series

As stated in Subsection 2.2.3, standard Bloom filters and standard cuckoo filters have a limited capacity. It is also true for our conflict-free replicated probabilistic filters. To allow filters to scale up as the number of elements increases, we propose a CRDT container named scalable filter series (SFS). Based on SFS, we design the scalable versions of our three CRPFs.

3.5.1 Supported Operations

Specification 5 demonstrates our SFS. It is made up of a list of homogeneous sub-filters implementing Specification 1. Membership queries are carried out as an aggregation over the list by checking if any sub-filter contains the queried element.

Apart from *query*, the SFS supports two types of *update* operations; one expands the series, and the other shrinks it. The *expand* (as well as *forceExpand*) operation appends an additional sub-filter to the series if the last one reaches its capacity limit, using the algorithm proposed by [1]. Operation *shrink* removes the last sub-filter from the series when it is

Specification 5: State-based scalable filter series

payload L: list⟨CRPF⟩
initial $L \leftarrow$ empty list
query *contains(e)*:
    return $\exists f \in L\colon f.contains(e)$
end
update *expand()*:
    if L *is empty* $\vee$ $L.last.cardinality \geq L.last.capacity$ **then**
        return $forceExpand()$
    else
        return L
    end
end
update *forceExpand()*:
    if L *is empty* **then**
        $f \leftarrow$ a new sub-filter whose FPR halves the expected compounded FPR
    else
        $f \leftarrow$ a new sub-filter whose FPR halves its predecessor's
    end
    return $L.append(f)$
end
update *shrink()*:
    if L *is empty* $\vee$ $L.last.cardinality > 0$ **then**
        return L
    else
        $L' \leftarrow$ remove the last sub-filter from L's list
        return $L'.shrink()$
    end
end
compare (L'):
    $w \leftarrow |L|$
    $w' \leftarrow |L'|$
    return $w \leq w' \wedge \forall i \in [0,w)\colon L[i].compare(L'[i])$
end
merge (L'):
    $w \leftarrow |L|$
    $w' \leftarrow |L'|$
    if $w > w'$ **then**
        return $L'.merge(L)$
    else
        return $L'[i \mapsto L'[i].merge(L[i])]_{i \in [0,w)}$
    end
end

empty for the purpose of conserving memory. Note that the cardinality of the Bloom filter and also GBF can be estimated with an algorithm proposed by [41]:

$$\text{cardinality} \approx -\frac{m}{k} \cdot \ln(1 - \frac{|S|}{m}). \tag{3.20}$$

The cardinality of the cuckoo filter, including GCF and ORCF, equals the number of entries it stores in the cuckoo hash table.

3.5.2 State Convergence

The *compare* and *merge* functions of SFS are defined as sub-filter-wise operations. If every sub-filter in one SFS is smaller than that at the same index in another SFS, we consider the former SFS to be smaller overall. Similarly, *merge* produces a new SFS where each sub-filter is the result of merging the corresponding elements from the two original SFSes. Figure 3.5 shows an example.

If the two operands of *compare* or *merge* have different lengths, the shorter one is conceptually padded with sub-filters in the bottom state. It is worth noting that the bottom state does not necessarily exist in the join semilattice of a state-based CRDT, and even if it does, it might not be used as the initial state. Nonetheless, the initial state of all our CRPFs is indeed the bottom $\varnothing$.

3.5.3 Probabilistic Characteristics

Since we make use of the exact scaling mechanism as [1], the FPR is also theoretically bounded by twice the FPR of the first sub-filter, which has been discussed in Subsection 2.2.3.

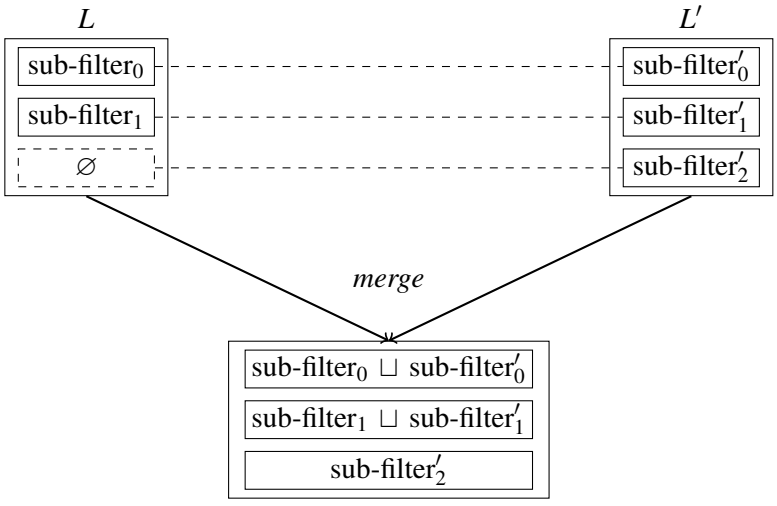


Figure 3.5: An example of merging two SFSes of different lengths. The shorter one, L, is conceptually padded with empty sub-filters. The two are then merged sub-filter-wise.

3.5.4 Integration

In this subsection, we put forward the scalable versions of the CRPFs in this chapter, namely, scalable grow-only Bloom filter (ScGBF), scalable grow-only cuckoo filter (ScGCF), and scalable observed-remove cuckoo filter (ScORCF). SFSes are integrated into them as payloads.

Specification 6 represents the specification of a state-based ScGBF. Unlike its non-scalable version, an element is added to the last available sub-filter only if it is not yet contained. In this way, we avoid adding the same element into multiple sub-filters. *contains*, *compare*, and *merge* are simply forwarded to the SFS and are thus omitted from the specification.

Adding elements to a state-based ScGCF, shown in Specification 7, differs from the procedure of the ScGBF in the condition to expand the series. Because the cuckoo hash table may report as being full before reaching its capacity, the ScGCF needs to scale in case of insertion failure. We leave out trivial forwarding operations for the same reason mentioned earlier.

We present the state-based ScORCF in Specification 8. Different from the two grow-only filters above, the ScORCF tries adding an element to not only the last sub-filter but also other non-full ones. Moreover, it synchronises the causal history with all sub-filters using the latest one and supplies it to newly created sub-filters. This is the reason for the additional field in the payload. The causal history is also involved in determining the partial order and least upper bound of the ScORCF. The *remove* operation is straightforward. It randomly selects a sub-filter that contains the element and requests the sub-filter to delete an entry.

Specification 6: State-based scalable grow-only Bloom filter

```
1  payload L: SFS⟨GBF⟩
2  initial L ← empty
3  update add(e):
4      if contains(e) then
5          return L
6      else
7          L′ ← L.expand()
8          i ← |L′| − 1 // the last
9          return L′[i ↦ L′[i].add(e)]
10     end
11 end
```

Specification 7: State-based scalable grow-only cuckoo filter

```
1  payload L: SFS⟨GCF⟩
2  initial L ← empty
3  update add(e):
4      if contains(e) then
5          return L
6      else
7          L′ ← L.expand()
8          try
9              i ← |L′| − 1 // the last
10             return L′[i ↦ L′[i].add(e)]
11         catch "maximum number of iterations is reached" then
12             return L′.forceExpand().add(e)
13         end
14     end
15 end
```

Specification 8: State-based scalable observed-remove cuckoo filter

payload L: SFS⟨ORCF⟩, H: causal history
initial $L \leftarrow$ empty, $H \leftarrow$ empty
query *contains(e)*:
    return $L.contains(e)$
end
update *add(e)*:
    $L' \leftarrow L.expand()$
    $i \leftarrow$ index of the last sub-filter $f \in L'$ such that $(f.cardinality < f.capacity)$ and $f.add(e)$ is successful
    if *such i exists* **then**
        $L'' \leftarrow L'[i \mapsto L'[i].add(e)]$
        $H'' \leftarrow L''[i].H$
        `// Keep causal history in sync`
        return $L''[j \mapsto (L''[j].S,\ H'')]_{j \in [0,|L''|)},\ H''$
    else
        return $L'.forceExpand().add(e)$
    end
end
update *remove(e)*:
    $R \leftarrow \{i \mid L[i].contains(e)\}$
    if $|R| = 0$ **then**
        return $L,\ H$
    else
        $i \leftarrow randomChoose(R)$
        $L' \leftarrow L[i \mapsto L[i].remove(e)]$
        return $L'.shrink(),\ H$
    end
end
compare *((L′, H′))*:
    return $L.compare(L') \wedge H.compare(H')$
end
merge *((L′, H′))*:
    return $L.merge(L'),\ H.merge(H')$
end

Chapter 4

Verification

In this chapter, we try to verify the correctness of the specifications in the previous chapter. We start by providing the formal definition of state-based CRDTs. Then, we verify using mathematical reasoning and an automatic prover.

4.1 Preliminaries

4.1.1 Definition of State-Based CRDT

A state-based CRDT, also known as a *convergent replicated data type* (CvRDT), is formally defined as a state-based object fulfilling the following three requirements according to [38].

1. Its payload state s is taken from a join semilattice $\langle S, \leq \rangle$ where the partial order $\leq$ is given by *compare*.
2. *merge* of any two states s_1, s_2 produces their least upper bound $s_1 \sqcup s_2$ in the semilattice.
3. Any *update* is an inflation that never decreases the state, i.e. $s \leq update(s)$.

4.1.2 Definition and Properties of Semilattice

A *semilattice* can be defined in either of the two equivalent ways: as a partially ordered set or as an algebraic structure [5, 12].

A *partially ordered set*, or *poset*, is a set S equipped with a binary relation $\leq$ such that:

$$\forall x \in S\colon x \leq x, \qquad \text{(reflexivity)} \tag{4.1}$$

$$\forall x, y \in S\colon x \leq y \wedge y \leq x \implies x = y, \qquad \text{(antisymmetry)} \tag{4.2}$$

$$\forall x, y, z \in S\colon x \leq y \wedge y \leq z \implies x \leq z. \qquad \text{(transitivity)} \tag{4.3}$$

The $\leq$ relation exhibiting the three properties above becomes the *partial order* on S. Given this partial order, an *upper bound* of two elements $x, y \in S$ is defined as the element $w \in S$ that $x \leq w \wedge y \leq w$. Among all such upper bounds, if there is one m that satisfies:

$$m \in S \wedge x \leq m \wedge y \leq m \wedge (\forall w\colon w \in S \wedge x \leq w \wedge y \leq w \implies m \leq w), \tag{4.4}$$

then m is called the *least upper bound*, also known as the *join*, of x and y, denoted as $x \sqcup y$. On top of poset, the definition of join semilattice can be drawn. If $\langle S, \leq \rangle$ is a poset and $\forall x, y \in S\colon x \sqcup y$ exists, then $\langle S, \leq \rangle$ is a join semilattice.

Viewed from an algebraic perspective, a join semilattice can be regarded as a set S equipped with a binary operation $\sqcup$. If the $\sqcup$ operation fulfils:

$$\forall x \in S\colon x \sqcup x = x, \qquad \text{(idempotence)} \tag{4.5}$$

$$\forall x, y \in S\colon x \sqcup y = y \sqcup x, \qquad \text{(commutativity)} \tag{4.6}$$

$$\forall x, y, z \in S\colon (x \sqcup y) \sqcup z = x \sqcup (y \sqcup z), \qquad \text{(associativity)} \tag{4.7}$$

then the algebraic structure $\langle S, \sqcup \rangle$ is proved to be a join semilattice whose partial order $\leq$ is derived as:

$$x \sqcup y = y \implies x \leq y. \tag{4.8}$$

Note again that the two forms of definition are exactly equivalent [12]. The $\leq$ that derives from $\sqcup$ of an algebraic structure leads to the same $\sqcup$ on poset, and vice versa.

4.2 Proof Sketch

In this section, we sketch out the proof of correctness of our CRPFs. These sketches are intended to outline the design rationale behind the operations. The formal verification is delayed to the next section.

We make use of the following lemma that declares a basic join semilattice of sets.

Lemma 4.1. *The power set of a set whose elements are partially ordered by set inclusion $\subseteq$ forms a join semilattice. The least upper bound operation is defined as a set union $\cup$.*

4.2.1 Correctness of Grow-Only Bloom Filter

Theorem 4.2. *GBF defined in Specification 2 is a valid CvRDT.*

Proof. Given its *compare* and *merge*, GBF is a join semilattice by definition. Moreover, *add* is a set union of the state and some other set; that is to say, the sole *update* is an inflation. Therefore, GBF is a CvRDT. □

4.2.2 Correctness of Grow-Only Cuckoo Filter

Recall that, in GCF, *compare* is defined with respect to the universe sets as Equation 3.4, yet *merge* in Equation 3.6 does not rely on them directly. We want to demonstrate first that our custom *merge* function is equivalent to a set union of the universe sets.

Lemma 4.3. *The dual cuckoo hash table operator D has the following properties:*

1. $D(D(S)) = S$
2. $D(S \cup T) = D(S) \cup D(T)$

3. $D(S \setminus T) = D(S) \setminus D(T)$

Proof.

1. D is an involution, because the internal *altIndexOf* is an involution.
2. A function preserves unions [33].
3. An injection preserves set differences [33]. D is involutory and is thus also injective.

□

Theorem 4.4. *The merge operation of GCF in Equation 3.6 is equivalent to the union of universes $U(S) \cup U(S')$.*

Proof. Note that state equivalence is defined in terms of the partial order, meaning that two states of the GCF are *equivalent* if and only if their universes are *identical*.

$$
\begin{aligned}
& U\big(S \cup (S' \setminus D(S))\big) \\
= \;& S \cup (S' \setminus D(S)) \cup D\big(S \cup (S' \setminus D(S))\big) \\
= \;& S \cup (S' \setminus D(S)) \cup D(S) \cup D(S' \setminus D(S)) \\
= \;& S \cup S' \cup D(S) \cup D(S' \setminus D(S)) \\
= \;& S \cup S' \cup D(S) \cup (D(S') \setminus S) \\
= \;& S \cup S' \cup D(S) \cup D(S') \\
= \;& S \cup D(S) \cup S' \cup D(S') \cup D(S) \cup S \cup D(S') \cup S' \\
= \;& \big(S \cup D(S) \cup S' \cup D(S')\big) \cup D\big(S \cup D(S) \cup S' \cup D(S')\big) \\
= \;& U\big(S \cup D(S) \cup S' \cup D(S')\big) \\
= \;& U\big(U(S) \cup U(S')\big)
\end{aligned}
$$

Therefore, $S \cup (S' \setminus D(S))$ and $U(S) \cup U(S')$ are equivalent. □

Apart from the theorem above, we still need two additional building blocks to prove the correctness of GCF; one concerns the *update* operation, and the other is about the *query* operation.

Theorem 4.5. *The add operation of GCF inflates the universe set $U(S)$.*

Proof. Firstly, we notice that a successfully added element ends up being a new entry in the table regardless of all possible displacement. Existing entries might be moved to their alternative buckets but never leave the universe. If the insertion fails or the element already presents, the state is not affected. To sum up, *add* always reaches a state that is at least as large as the original one in terms of set inclusion. And, since a function preserves inclusions [33], *add* inflates the universe. □

Theorem 4.6. *The contains operation of GCF returns the same result for a state S and its universe $U(S)$.*

Proof. The *contains* operation looks for the matching entry in both candidate buckets, going over both S and $D(S)$. Since $U(S)$ is the union of the two sets, checking whether an entry is included in $U(S)$ gives the same result as doing so with S. □

We are now able to show that GCF behaves the same way as CvRDT.

Theorem 4.7. *GCF defined in Specification 3 is a valid CvRDT.*

Proof. Given a GCF with state S, consider a companion state-based object whose payload is the corresponding universe set $U(S)$. We define its *compare* as ordinary set inclusion and *merge* as set union. The *query* and *update* operations are the same as those of the GCF. According to Lemma 4.1, the possible states of the companion object form a join semilattice. Furthermore, because all *update* operations inflate $U(S)$, the companion object becomes a CvRDT.

Comparing GCF with the companion object, we observe that all *query* operations yield the same values on them. For this reason, we consider their states as equivalent [38]. In addition, they adopt identical partial order and produce equivalent upper bounds. Thus, they are regarded as equivalent in general. Since the companion object is a CvRDT, GCF is also a CvRDT. □

4.2.3 Correctness of Observed-Remove Cuckoo Filter

To simplify the proof, we leverage the fact that the optimised ORSet in [4] is known to be a CvRDT.

Lemma 4.8. *ORSet is a valid CvRDT.*

Reviewing the specification of ORSet, we notice that whether or not an entry is preserved in the merged state solely depends on the existence of its version tag in the two merging states. A version tag is kept if either of the following is true.

1. It is in both sets of added elements of the two merging states.
2. It is in the set of added elements of one state and is not observed by the other state.

Moreover, the partial order is also related to version tags only. A smaller state cannot include more version tags in the casual history or remove more from the set of added elements. In other words, a smaller state has a smaller casual history and a smaller (conceptual) set of tombstones.

We now show that ORCF uses an identical approach to arbitrate elements. Recall that we divide the entries of ORCF into four categories, covered in Equation 3.12 to Equation 3.15. S_1 and S_4 correspond to the first case in the ORSet, while S_2 and S_3 are the same as the second case. Meanwhile, it is obvious that the *compare* function of ORCF is identical to that of ORSet. Since ORCF manages version tags the same way as ORSet, the states of an ORCF also make up a join semilattice.

In addition, the two *updates* are both inflations. *add* inserts a new version tag into the causal history, and *remove* inserts one into the tombstone set. As a result, we have proved the following theorem.

Theorem 4.9. *ORCF defined in Specification 4 is a valid CvRDT.*

4.3 Verifying with VeriFx

Despite the numerous existing CRDTs, their correctness is difficult to verify [21, 28]. Formal paper proofs are tedious and prone to mistakes [43, 21]. VeriFx [14] is proposed as a tool offering automatic verification of the correctness of CRDT implementations. It translates Scala-like high-level code into Z3 programme and then verifies with the Z3 theorem prover [13]. In the bundled example proof, the algebraic structure approach is used, partially though. That is, it tries proving whether *merge* is idempotent, commutative, and associative, using Equation 4.5, Equation 4.6, and Equation 4.7. If all three yield positive results, the CvRDT is regarded as correct [14].

4.3.1 Modifications

There is a missing condition to check in the default proof: Equation 4.8. Although a *merge* with the three properties above does define a join semilattice, the semilattice may derive a partial order different from the one defined by *compare*. This is because the three steps above do not utilise *compare* directly, but instead only make use of the equivalence relation defined in terms of *compare* as Equation 4.2. As a result, any different partial order that gives the same equivalence relation is able to pass the verification. Table 4.1 demonstrates some erroneous CvRDTs with such configurations. Despite their mismatched *merge* and *compare*, all of them are considered correct by the bundled example proof. Yet, even after passing this check, the verification process is still only halfway done. All *update* operations need to be verified for being inflation.

Thus, we modify the bundled proof so that faulty implementations can be identified. The new proof strictly complies with the definition of join semilattice as an algebraic structure defined by Equation 4.5 to Equation 4.8. Besides, we also include a proof using the poset

Table 4.1: Examples of erroneous CvRDTs yet still pass the verification.

Payload S	User-defined $x.merge(y)$	*merge*-derived $x.compare(y)$	User-defined $x.compare(y)$
$\mathbb{Z}$	$max(x,y)$	$x \leq y$	$y \leq x$
$\mathbb{Z}$	$min(x,y)$	$x \geq y$	$y \geq x$
$\mathscr{P}(\mathbb{Z})$	$x \cup y$	$x \subseteq y$	$y \subseteq x$
$\mathscr{P}(\mathbb{Z})$	$x \cap y$	$x \supseteq y$	$y \supseteq x$
$\mathscr{P}(\mathbb{Z})$	$\begin{cases} y, & (x = y) \\ \varnothing, & (x \neq y) \end{cases}$	$x = y \vee y = \varnothing$	$y \supseteq x$

approach, corresponding to Equation 4.1 to Equation 4.4. Either one may be used while the other serves as a double-check since they are equivalent. In addition, we leave a comment to remind programmers to incorporate proofs of their *update* operations themselves since *update* operations vary a lot and sharing code might introduce unnecessary complexity.

4.3.2 Results

We implement all specifications mentioned in Chapter 3 using VeriFx. All of them successfully pass the modified verification. Therefore, we are convinced that our state-based conflict-free replicated probabilistic filter specifications are correct.

Chapter 5

Implementation

We implement our conflict-free replicated probabilistic filters as a Java/Scala library. The library is open-source and publicly accessible on our GitHub repository. In this chapter, we point out a few parts of the implementation that are worth noting.

5.1 Immutable and Mutable Variants

We implement both immutable and mutable versions of our conflict-free replicated probabilistic filters. They have the same API and are mostly implemented in the same way, except that the immutable variants return new instances after being "updated" while the mutable variants are updated in place and return themselves.

There is one exception regarding the SFS. Specification 5 implies that only the overlapping part of two SFSes need to be explicitly merged, while the exceeding sub-filters can be shared directly. However, it only works for immutable sub-filters. In the case of mutable ones, we still need to merge them with button states, essentially making a copy.

5.2 Hash Functions Used

We utilise `MurmurHash3` and `FarmHash` provided by the Google Guava project [22] in favour of their performance and low collision rate. In conflict-free replicated Bloom filters, the two are used as the bases to generate the series of hash codes following Kirsch-Mitzenmacher optimisation [26]. In conflict-free replicated cuckoo filters, the former acts as the index hash function, and the latter is the fingerprint hash function.

5.3 Encoding of Cuckoo Entries

In GCF, entries are simple fingerprints without any special encoding. A `byte` is allocated to store a fingerprint that is at most 8 bits long, and a `short` for a fingerprint up to 16 bits. In ORCF, entries are not directly stored as tuples of three integers. Instead, they are encoded as primitive types. Two configurations are provided. The first one uses a 32-bit `int` per entry, which consists of an 8-bit fingerprint, a 4-bit replica ID, and a 20-bit timestamp. The other

encodes an entry as `long`, reserving 16 bits for the fingerprint, 16 bits for the replica ID, and 32 bits for the timestamp. The restrictions on length are merely limitations in practice rather than in theory.

Chapter 6

Evaluation

This chapter describes the experimental evaluation of the conflict-free replicated probabilistic filters. Our evaluation tests for the following properties.

1. The empirical FPR of CRPFs compared to the theoretical value and that of ordinary filters.
2. The memory consumption of CRPFs compared to CRDT sets.

6.1 General Experiment Setup

We evaluate CRPFs configured as the following. The non-scalable filters are always created with capacity matching the quantity of workload. The scalable ones start with a sub-filter whose capacity is a quarter of the total quantity of workload. Subsequent sub-filters use the same capacity without further expansion. Cuckoo filters are all configured with an iteration quota of 500 attempts [17]. The names of the configurations are listed below.

- `GBF 5`: GBF having 5 hash functions
- `GCF 4`: GCF whose bucket expects 4 entries with 8-bit fingerprint
- `ORCF 4`: ORCF whose bucket expects 4 entries with 8-bit fingerprint
- `ScGBF 5`: ScGBF whose compounded FPR is the same as `GBF 5`
- `ScGCF 4`: ScGCF whose compounded FPR is the same as `GCF 4`
- `ScORCF 4`: ScORCF whose compounded FPR is the same as `ORCF 4`

The parameters for `GBF 5`, `GCF 4`, and `ORCF 4` are picked so that their theoretical FPR are all 3.125%.

As for workload generation, we adopt an approach similar to previous works on probabilistic filters [17, 8, 23]. We use as keys 128-bit random integers generated by the standard library of Java. The workload can be divided into parts to be executed on different replicas. We only distribute it to at most two replicas in the experiments. 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. The label *all local* denotes the configuration where the workload is handled by a single replica locally.

The workload may contain two types of *update* operations: *add* and *remove*. We insert all elements in the *add* workload into Bloom filters and sets. However, we stop inserting into cuckoo filters when the maximum number of iterations is reached, i.e. when they are full [17]. *removes* are only generated for elements that the filter or set already contains. We denote the workload pattern containing $a\%$ of *add* and $(1-a\%)$ of *remove* as an *$a\%$-add workload*.

6.2 Empirical False Positive Rate

6.2.1 Setup

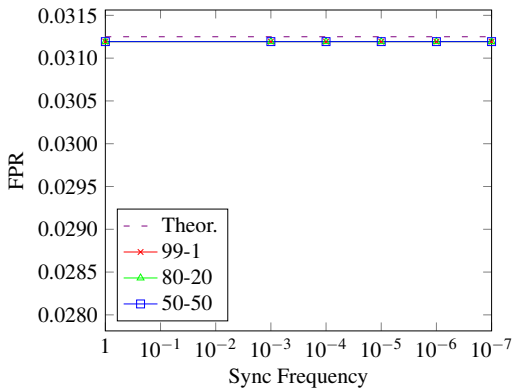
In this section, we measure the empirical FPR of different CRPFs concerning different workload distributions and sync frequencies. We create two replicas of the studied filter, apply 2^{20} *adds* split according to a chosen distribution scheme, and merge them every 10^3 to 10^7 operations. Additionally, there is always a final *merge* after all *adds* are done. Three representative distribution schemes are used: 50-50 split, 80-20 split, and 99-1 split. For reference, a pair of fully synchronised CRPFs are also evaluated. The result is regarded as the FPR of the corresponding ordinary filter.

The empirical FPR is estimated by testing 2^{20} random keys that are never inserted and checking if they are reported as being included. The procedure is repeated five times using different keys and testing datasets. The average values of the metrics are recorded, shown in Figure 6.1.

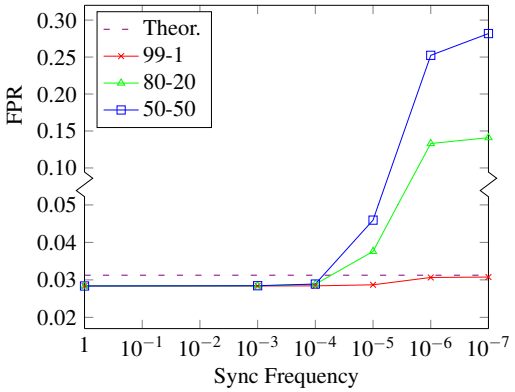
6.2.2 Results

Figure 6.1a illustrates the FPR of `GBF 5`. It shows that the value is identical to the regular Bloom filter and is close to the theoretical FPR. Moreover, the FPR is unaffected by the distribution scheme or sync frequency because of the simple design of *merge*. The merged state, being a set union, is always identical to the state resulting from local *adds*. Therefore, we consider GBF a great drop-in replacement for the Bloom filter in distributed settings.

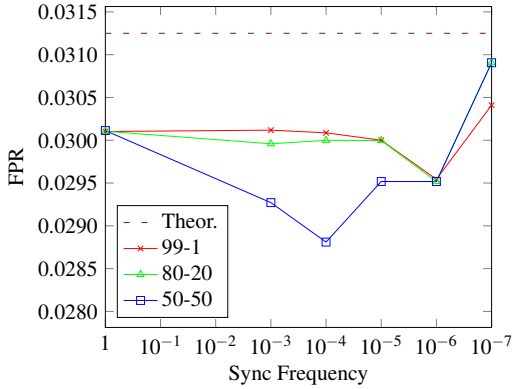
From Figure 6.1b, we observe that the FPR of `GCF 4` has mixed performance according to the workload distribution and the sync frequency. When the filters are merged relatively frequently, the FPR drops as the distribution becomes more even. We believe that this phenomenon originates from the counterbalancing nature of *add* and *merge*. *merge* is inclined to introduce overflowing buckets to accommodate all the entries in the two merging filters. Conversely, *add* is forbidden from producing more overflowing buckets and actively tries to reduce the number of overflowing buckets. In case of an even split, both GCFs have a considerable number of full buckets before *merge*. It leads to more overflowing buckets in the merged state, which makes subsequent *adds* more likely to fail. The filter thus reports a lower load factor and therefore a lower FPR.



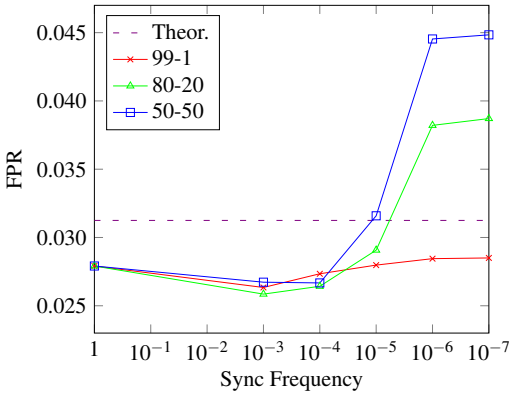
(a) GBF 5



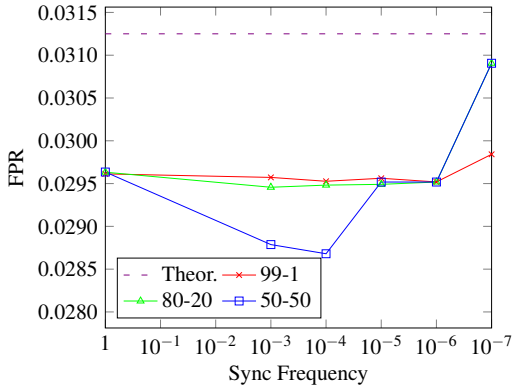
(d) ScGBF 5



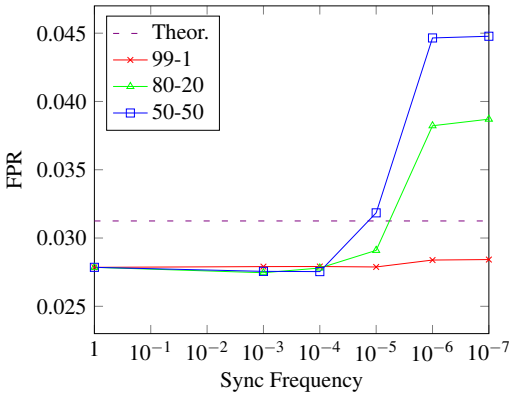
(b) GCF 4



(e) ScGCF 4



(c) ORCF 4



(f) ScORCF 4

Figure 6.1: Effect of workload distribution and sync frequency on empirical FPR

However, if the filters are barely merged, the relation between those factors inverts. This is due to the uptick in the load factor. A cuckoo filter with four slots per bucket can be filled to a maximum load factor of 95% [17]. If the workload is split into two parts evenly, each filter is able to buffer its part before *merge*. The merged filter thus reaches a higher load factor, which results in a higher FPR. As a consequence, the FPR first decreases and then increases as the sync frequency decreases.

The analysis above also holds for `ORCF 4`, whose results are shown in Figure 6.1c.

Unfortunately, the empirical FPRs of scalable CRPFs diverge dramatically from the expected values, depicted in Figure 6.1d, Figure 6.1e, and Figure 6.1f. We notice that the values deteriorate much more rapidly than those in the non-scalable CRPFs. The surge stems from the biased distribution of elements within the SFS of a scalable CRPF. When the sync frequency is low, each replica places the elements in the first few sub-filters. Those sub-filters are sufficient for accommodating the assigned workload and are full already before *merge*. As a result, they contain more elements than expected after *merge*, which leads to the worsening of FPR. We argue that scalable CRPFs typically require a relatively high frequency of *merge* to maintain the expected FPR.

6.3 Memory Consumption

6.3.1 Setup

We also evaluate the space efficiency of the CRPFs. The metric used is the serialised size, which is the number of bytes for network transmission. We measure the raw size and the size after GZIP compression [15], a common compression scheme for web protocols. The configurations used are `GBF 5`, `GCF 4`, and `ORCF 4`. For comparison, we make use of the `GSet` and `ORSet` implemented by Akka [30]. The CRDTs in this experiment are loaded with *a*%-add workload containing 2^{20} operations, where *a* can be 100, 80, or 51. The workload is either split evenly between two replicas (50-50 split) or simply executed locally on a single replica (all local). In case of a split, the CRDTs are merged after all operations are finished. The experiment repeats five times, and we take the average values as results.

6.3.2 Results

Table 6.1 compares the serialised size of the grow-only CRDTs: `GBF 5`, `GCF 4`, and `GSet`. It clearly demonstrates the space benefits of the CRPFs. Both `GBF 5` and `GCF 4` are serialised into much less data than `GSet` without sacrificing much load factor. Note that there is an uptick in the raw serialised size of `GCF 4` under 50-50 split. This is due to the large amount of overflowing buckets which store approximately 17% of the entries. These buckets are implemented as a map from indexes to byte arrays, which is serialised much less efficiently than the main buckets backed by an array. The discrepancy becomes smaller after compression.

`ORCF 4` is compared with `ORSet` under various configurations of *add* ratio, depicted in Table 6.2. Our `ORCF 4` utilises a relatively constant amount of memory. On the other hand, `ORSet` assigns a fixed number of bytes for each element, and its size depends mainly on the

load of the set. When the load is high, `ORSet` consumes more memory than `ORCF 4`. When the load is low, it does require less space than `ORCF 4`. However, in such cases, `ORCF 4` mainly consists of 0, and can thus be compressed into a smaller object.

Table 6.1: Serialised size of grow-only filters and grow-only sets. BPE stands for *bytes per element*.

Workload	CRDT	Load Factor	Serialised Size (MB)	Serialised BPE	Compressed Size (MB)	Compressed BPE
100% add all local	`GBF 5`	100%	1.01	1.01	**0.91**	0.91
	`GCF 4`	96%	1.05	1.05	**1.04**	1.04
	`GSet`	100%	23.07	22.00	17.16	16.37
100% add 50-50 split	`GBF 5`	100%	1.01	1.01	**0.91**	0.91
	`GCF 4`	98%	3.73	3.62	**1.59**	1.54
	`GSet`	100%	23.07	22.00	17.16	16.37

Table 6.2: Serialised size of observed-remove filters and sets.

Workload	CRDT	Load Factor	Serialised Size (MB)	Serialised BPE	Compressed Size (MB)	Compressed BPE
100% add all local	`ORCF 4`	96%	8.39	8.37	**4.75**	4.74
	`ORSet`	100%	51.38	49.00	21.16	20.18
80% add all local	`ORCF 4`	60%	8.39	13.34	**3.42**	5.44
	`ORSet`	60%	30.83	49.00	12.66	20.13
51% add all local	`ORCF 4`	2%	8.39	400.14	**0.19**	9.16
	`ORSet`	2%	1.03	49.00	0.44	20.87
100% add 50-50 split	`ORCF 4`	100%	12.54	11.96	**5.71**	5.45
	`ORSet`	100%	51.38	49.00	21.45	20.46
80% add 50-50 split	`ORCF 4`	60%	9.25	14.70	**3.54**	5.62
	`ORSet`	60%	30.83	49.00	12.86	20.44
51% add 50-50 split	`ORCF 4`	2%	8.39	381.81	**0.20**	9.10
	`ORSet`	2%	1.03	49.00	0.45	21.25

Chapter 7

Related Work

This chapter reviews existing literature that uses probabilistic filters in CRDTs. It also shows how they are related to yet different from the work in this thesis.

[18] proposes Probabilistic Causal Context, a substitution for classic deterministic causal context. Being a causal context, it stores causality metadata about the versions known by the CRDT it is part of. Meanwhile, its size does not grow linearly with the number of nodes due to its probabilistic approach. At its core is an age-partitioned Bloom filter [40]. This variant of the Bloom filter represents, probabilistically, a sliding window over a stream of elements. Old elements are forcibly removed from the filter in order to make room for new data. As a result, Probabilistic Causal Context produces both false positives and false negatives. When used as the causal context of a causal CRDT, e.g. an ORSet, the Probabilistic Causal Context leads to inconsistencies of both types in the CRDT.

BloomCRDT, designed by [7], is a CRDT that provides the semantics of the ORSet probabilistically. In contrast to traditional ORSet which uses a set to record tombstones, BloomCRDT uses a Bloom filter as an approximation to the tombstone set. However, false positives in the tombstone may lead to faulty removal of elements, that is, false negatives in BloomCRDT.

Unlike the previous literature, the CRPFs in this thesis may only result in false positives and never false negatives. In addition, our CRPFs function independently as probabilistic filters rather than as parts of other CRDTs.

Chapter 8

Conclusion

This chapter concludes the thesis and addresses the research questions raised in Section 1.1. Following that, some ideas for future work are discussed.

8.1 Conclusions

In this thesis, we propose three concrete designs for conflict-free replicated probabilistic filters for approximate membership queries in distributed systems. We also extend scalable filters into the context of CRDT and put forward the scalable variants of the three CRPFs above. The thesis elaborates on the specifications of these CRPFs and verifies their correctness. Furthermore, we empirically analyse their FPR and memory consumption.

In conclusion, our CRPFs are proved to be valid CRDTs. We find that their empirical FPR aligns well with the theoretical value when the CRPFs are merged relatively frequently. They consume less memory compared to CRDT sets regardless of workload. Our CRPFs are indeed a solution to distributed AMQ.

8.1.1 Answers to Research Questions

- **RQ 1**: What operations does CRPF support?

  ANS 1: We define an interface of CRPF in Specification 1. The *contains*, *add*, and *remove* operations offer ways to manipulate the CRPF as a probabilistic filter. The *compare* and *merge* functions make it a CRDT. Detailed designs for the CRPFs are elaborated in Chapter 3. We prove their correctness using paper proof and software verification in Chapter 4.

- **RQ 2**: What are the false positive rate and false negative rate of CRPF?

  ANS 2: In Chapter 3, we derive formulae for estimating the false positive rate of the CRPFs. We propose a requirement on *remove* operations to keep the false negative rate as zero.

- **RQ 3**: How does CRPF perform empirically?

ANS 3: We conduct experiments to analyse the empirical FPR and memory consumption of the CRPFs. The results shown in Chapter 6 demonstrate that FPR can be affected by workload distribution and the sync frequency. Furthermore, CRPFs generally require less space than CRDT sets, thanks to that they store only hash representations.

8.2 Future Work

We identify the following directions for future investigation.

Better analyses of the probabilistic characteristics. The current algorithm to remove elements from ORCF *might* introduce false positives as well as false negatives. However, the formula for FPR is a rough estimation. A better analysis can be drawn if more information about the operation history is taken into account. Similarly, false negatives also depend on the operation history. We only find that the false negative rate is zero when all removals are causally-safe. The precise probability in other circumstances is unexplored.

δ-based replication. The specifications of CRPFs in this thesis are all state-based, meaning that the whole state will be transferred upon merge. Despite experiments demonstrating that the size of the state is relatively small, one can still argue that the size is not negligible. The transferred state can be further reduced by adopting δ-based replication [2]. These CRDTs are called δ-CRDTs. They disseminate the changes δ rather than sending the whole states so that network overhead can be optimised. Implementing efficient δ-based CRPFs is left as a future work.

Other probabilistic and concurrency semantics. It is also interesting to design CRDTs based on other probabilistic filters, such as the counting Bloom filter [10], the quotient filter [3], and the Morton filter [8]. Concurrency semantics other than grow-only and observed-remove, like last-writer-wins, can also be applied to CRPFs. Those semantics are useful in specialised settings [38].

Bibliography

[1] Paulo Sérgio Almeida, Carlos Baquero, Nuno Preguiça, and David Hutchison. Scalable bloom filters. *Information Processing Letters*, 101(6):255–261, 2007.

[2] Paulo Sérgio Almeida, Ali Shoker, and Carlos Baquero. Efficient state-based crdts by delta-mutation. In *International Conference on Networked Systems*, pages 62–76. Springer, 2015.

[3] Michael A Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P Spillane, and Erez Zadok. Don't thrash: How to cache your hash on flash. *Proceedings of the VLDB Endowment*, 5(11), 2012.

[4] Annette Bieniusa, Marek Zawirski, Nuno Preguiça, Marc Shapiro, Carlos Baquero, Valter Balegas, and Sérgio Duarte. An optimized conflict-free replicated set. *arXiv preprint arXiv:1210.3368*, 2012.

[5] Garrett Birkhoff. *Lattice theory*, volume 25. American Mathematical Soc., 1940.

[6] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[7] Ewout Bongers. Conflict free r tree. Master's thesis, Delft University of Technology, 2021.

[8] Alex D Breslow and Nuwan S Jayasena. Morton filters: faster, space-efficient cuckoo filters via biasing, compression, and decoupled logical sparsity. *Proceedings of the VLDB Endowment*, 11(9):1041–1055, 2018.

[9] Eric A Brewer. Towards robust distributed systems. In *PODC*, volume 7, pages 343–477. Portland, OR, 2000.

[10] Andrei Broder and Michael Mitzenmacher. Network applications of bloom filters: A survey. *Internet mathematics*, 1(4):485–509, 2004.

[11] Russell Brown, Sean Cribbs, Christopher Meiklejohn, and Sam Elliott. Riak dt map: a composable, convergent replicated dictionary. In *Proceedings of the First Workshop on Principles and Practice of Eventual Consistency*, pages 1–1, 2014.

[12] Brian A Davey and Hilary A Priestley. *Introduction to lattices and order*. Cambridge university press, 2002.

[13] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[14] Kevin De Porre, Carla Ferreira, and Elisa Gonzalez Boix. Verifx: Correct replicated data types for the masses. In *European Conference on Object-Oriented Programming*, 2023.

[15] L Peter Deutsch. GZIP file format specification version 4.3. `https://rfc-editor.org/rfc/rfc1952.txt`, 1996. Accessed: May 31, 2024.

[16] Martin Dietzfelbinger and Rasmus Pagh. Succinct data structures for retrieval and approximate membership. In *International Colloquium on Automata, Languages, and Programming*, pages 385–396. Springer, 2008.

[17] Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 75–88, 2014.

[18] Pedro Henrique Fernandes and Carlos Baquero. Probabilistic causal contexts for scalable crdts. In *Proceedings of the 10th Workshop on Principles and Practice of Consistency for Distributed Data*, pages 1–8, 2023.

[19] Andrii Gakhov. *Probabilistic data structures and algorithms for big data applications*. BoD–Books on Demand, 2022.

[20] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *Acm Sigact News*, 33(2):51–59, 2002.

[21] Victor BF Gomes, Martin Kleppmann, Dominic P Mulligan, and Alastair R Beresford. Verifying strong eventual consistency in distributed systems. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):1–28, 2017.

[22] Google. Guava explained. `https://github.com/google/guava/wiki`, 2016. Accessed: May 31, 2024.

[23] Thomas Mueller Graf and Daniel Lemire. Xor filters: Faster and smaller than bloom and cuckoo filters. *Journal of Experimental Algorithmics (JEA)*, 25:1–16, 2020.

[24] Gaurav Gupta, Minghao Yan, Benjamin Coleman, RA Elworth, Tharun Medini, Todd Treangen, and Anshumali Shrivastava. RAMBO: Repeated and merged bloom filter for ultra-fast multiple set membership testing (MSMT) on large-scale data. *arXiv preprint arXiv:1910.02611*, 2019.

[25] Kun Huang and Tong Yang. Additive and subtractive cuckoo filters. In *2020 IEEE/ACM 28th International Symposium on Quality of Service (IWQoS)*, pages 1–10. IEEE, 2020.

[26] Adam Kirsch and Michael Mitzenmacher. Less hashing, same performance: Building a better bloom filter. In *Algorithms–ESA 2006: 14th Annual European Symposium, Zurich, Switzerland, September 11-13, 2006. Proceedings 14*, pages 456–467. Springer, 2006.

[27] Martin Kleppmann and Alastair R Beresford. A conflict-free replicated JSON datatype. *IEEE Transactions on Parallel and Distributed Systems*, 28(10):2733–2746, 2017.

[28] Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, and Joseph M Hellerstein. Katara: Synthesizing crdts with verified lifting. *Proceedings of the ACM on Programming Languages*, 6(OOPSLA2):1349–1377, 2022.

[29] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications*, 1978.

[30] Lightbend. Akka documentation. `https://doc.akka.io/`, 2011. Accessed: May 31, 2024.

[31] Dahlia Malkhi and Doug Terry. Concise version vectors in winfs. In *International Symposium on Distributed Computing*, pages 339–353. Springer, 2005.

[32] Stéphane Martin, Mehdi Ahmed-Nacer, and Pascal Urso. Abstract unordered and ordered trees crdt. *arXiv preprint arXiv:1201.1784*, 2012.

[33] James R Munkres. *Topology*. Prentice Hall, 2000.

[34] DS Parker, GJ Popek, G Rudisin, A Stoughton, BJ Walker, E Walton, JM Chow, D Edwards, S Kiser, and C Kline. Detection of mutual inconsistency in distributed systems. *IEEE Transactions on Software Engineering*, 9(03):240–247, 1983.

[35] Redis. HyperLogLog in active-active databases. `https://redis.io/docs/latest/operate/rs/databases/active-active/develop/data-types/hyperloglog/`, 2023. Accessed: Dec 14, 2023.

[36] Hyun-Gul Roh, Myeongjae Jeon, Jin-Soo Kim, and Joonwon Lee. Replicated abstract data types: Building blocks for collaborative applications. *Journal of Parallel and Distributed Computing*, 71(3):354–368, 2011.

[37] Mahmoud Sayrafiezadeh. The birthday problem revisited. *Mathematics Magazine*, 67 (3):220–223, 1994.

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

[39] Marc Shapiro, Nuno Preguiça, Carlos Baquero, and Marek Zawirski. Conflict-free replicated data types. In *Stabilization, Safety, and Security of Distributed Systems: 13th International Symposium, SSS 2011, Grenoble, France, October 10-12, 2011. Proceedings 13*, pages 386–400. Springer, 2011.

[40] Ariel Shtul, Carlos Baquero, and Paulo Sérgio Almeida. Age-partitioned bloom filters. *arXiv preprint arXiv:2001.03147*, 2020.

[41] S Joshua Swamidass and Pierre Baldi. Mathematical correction for fingerprint similarity measures to improve chemical retrieval. *Journal of chemical information and modeling*, 47(3):952–964, 2007.

[42] Dinusha Vatsalan and Peter Christen. Multi-party privacy-preserving record linkage using bloom filters. *arXiv preprint arXiv:1612.08835*, 2016.

[43] Peter Zeller, Annette Bieniusa, and Arnd Poetzsch-Heffter. Formal specification and verification of crdts. In *34th Formal Techniques for Networked and Distributed Systems (FORTE)*, pages 33–48. Springer, 2014.

[44] Yuqi Zhang, Lingzhi Ouyang, Yu Huang, and Xiaoxing Ma. Conflict-free replicated priority queue: Design, verification and evaluation. In *Proceedings of the 14th Asia-Pacific Symposium on Internetware*, pages 302–312, 2023.