Aleph Filter

To Infinity in Constant Time

Understanding, Analysis, and Rust Implementation

Neel Bansal

March 30, 2026

A companion technical report to the first Rust implementation of the Aleph Filter, based on the VLDB 2024 paper by Dayan, Bercea, and Pagh.

arxiv.org/abs/2404.04703 · github.com/NPX2218/aleph-filter

Contents

1 Background: Probabilistic Filters **1**
1.1 What Are Filters? 1
1.1.1 Hash Functions and Collisions 1
1.1.2 The Need for Filters 1
1.2 Bloom Filters 1
1.2.1 Structure and Operations 2
1.2.2 Worked Example 2
1.2.3 False Positive Rate 4
1.2.4 Space Efficiency 6
1.3 Cuckoo Filters 6
1.3.1 Structure and Operations 7
1.3.2 Worked Example 7
1.3.3 From Cuckoo Hashing to Cuckoo Filters 8
1.4 Summary of Filter Limitations 12

2 Quotient Filters and Expansion **14**
2.1 The Quotienting Idea 14
2.1.1 Worked Example: Insertion 15
2.2 Metadata Bits 15
2.2.1 Worked Example: Metadata in Action 16
2.2.2 Lookup Algorithm 17
2.3 How Expansion Works 17
2.3.1 The Expansion Procedure 18
2.4 The Void Entry Problem 19
2.4.1 How Void Entries Arise 19
2.4.2 The Expansion Dilemma 20
2.4.3 InfiniFilter's Approach: Secondary Tables 20
2.4.4 The Aleph Filter's Solution: Void Duplication 21

3 The Aleph Filter **23**
3.1 Overview 23
3.2 Structure 23
3.2.1 The Mother Hash 24
3.3 Void Entry Duplication 25
3.3.1 Worked Example 25
3.3.2 Why Duplication Is Correct 26
3.3.3 How Queries Find Void Entries 26
3.3.4 Why Duplication Doesn't Explode 26
3.4 Constant-Time Queries 27
3.5 Tombstone-Based Deletes 28
3.5.1 Phase 1: Immediate Tombstone ($\mathcal{O}(1)$) 28
3.5.2 Using the Mother Hash for Duplicate Removal 29
3.5.3 Phase 2: Deferred Duplicate Removal 29
3.5.4 Why Delete the Longest Matching Mother Hash? 30
3.5.5 Computational Cost 30
3.6 Rejuvenation 30
3.6.1 Rejuvenation Procedure 30

3.7 FPR Analysis . . . 31
3.7.1 Per-Generation FPR . . . 31
3.7.2 Total FPR . . . 32
3.7.3 Widening Regime . . . 32
3.8 Operating Regimes . . . 32
3.9 Architecture Summary . . . 33
3.10 Comparison . . . 33

4 Rust Implementation **34**
4.1 Data Structures . . . 34
4.2 Operations . . . 34
4.2.1 Insertion . . . 35
4.2.2 Query . . . 35
4.2.3 Expansion . . . 36
4.2.4 Deletion . . . 37
4.3 Fidelity to the Paper . . . 39
4.3.1 What Matches Exactly . . . 39
4.3.2 What Does Not Match . . . 39

5 Comparison and Conclusion **41**
5.1 Benchmark Methodology . . . 41
5.1.1 Experimental Setup . . . 41
5.1.2 Competing Implementations . . . 41
5.2 Feature Comparison . . . 42
5.3 Throughput Benchmarks . . . 42
5.3.1 Insertion Throughput . . . 42
5.3.2 Positive Lookup Throughput . . . 42
5.3.3 Negative Lookup Throughput . . . 42
5.3.4 Deletion Throughput . . . 43
5.4 False Positive Rate . . . 43
5.5 Expansion Performance . . . 44
5.6 Conclusion . . . 44
5.7 Future Work . . . 45

Chapter 1

Background: Probabilistic Filters

1.1 What Are Filters?

1.1.1 Hash Functions and Collisions

Let U denote the universe of all possible elements, and let $S \subseteq U$ with $n = |S|$ denote the set of elements we wish to represent.

A **hash function** is a deterministic function $h : U \to \{0, 1, \ldots, m-1\}$ that maps elements from a universe U (e.g., strings, integers) to a fixed range of m possible values. Determinism means that for any input $x \in U$, repeated evaluations of $h(x)$ always yield the same result.

Because the universe U is typically much larger than the range m (i.e., $|U| \gg m$), multiple distinct inputs can map to the same output. This phenomenon is called a **collision**: two elements $x \neq y$ such that $h(x) = h(y)$. By the pigeonhole principle, if $|U| > m$, collisions are unavoidable and no hash function can be injective over such a domain.

A "good" hash function distributes outputs approximately uniformly: for a randomly chosen $x \in U$, each value in $\{0, 1, \ldots, m-1\}$ should be roughly equally likely. This notion was formalized by *universal hashing* [1], which guarantees that for any $x \neq y$, the probability of collision satisfies $\Pr[h(x) = h(y)] \leq 1/m$. This property is essential for the data structures that follow.

1.1.2 The Need for Filters

Many systems store large datasets on slow media (e.g. disk, or across a network). A common operation is checking whether a key x exists in a set S before performing an expensive lookup. If we could cheaply determine that $x \notin S$, we could skip the lookup entirely.

Storing S exactly requires $\Omega(n \log |U|)$ bits for $n = |S|$ elements, which may be prohibitively large. **Filters** address this by trading exactness for space: they use far less memory by tolerating a small probability of error.

Probabilistic Filter

A **filter** is a compact data structure that represents a set S and answers membership queries: "Is element x in S?" It can return **false negatives: never**, but may return **false positives** with some bounded probability ϵ.

1.2 Bloom Filters

A Bloom filter, introduced by Burton Bloom in 1970 [2], is a probabilistic data structure for approximate set membership testing. Given a set S, a Bloom filter can answer queries of the form "Is $x \in S$?" with a one-sided error:

- If $x \in S$, the filter always returns YES (no false negatives).
- If $x \notin S$, the filter may incorrectly return YES with probability at most ϵ (a false positive).

This property made Bloom filters practical in real systems. For example, Google Chrome [3] used a Bloom filter until 2012 to check URLs against a local list of known malicious sites. If the filter returned NO, the URL was safe and no further lookup was needed. Only on a YES did the browser perform a more expensive server-side check.

1.2.1 Structure and Operations

A Bloom filter consists of two components

1. A **bit array** B of m bits, where:

$$B[i] \leftarrow 0 \quad \forall i \in \{0, 1, 2, \ldots, m-1\}$$

2. A family of k independent hash functions $h_1, h_2, \ldots, h_k$, each mapping elements from the universe U uniformly to $\{0, 1, \ldots, m-1\}$.

The Bloom filter also supports the following operations:

1. **Insertion:** To insert an element x into the filter, compute $h_1(x), h_2(x), \ldots, h_k(x)$ and set each corresponding bit to 1:

$$B[h_i(x)] \leftarrow 1 \quad \forall i \in \{1, 2 \ldots, k\}$$

2. **Query:** To test whether an element x is in the set, check whether all k bits are set:

$$\text{query}(x) = \bigwedge_{i=1}^{k} B[h_i(x)]$$

If any bit $B[h_i(x)] = 0$, then x was never inserted, this is certain because insertion only sets bits to 1, never back to 0. If all k bits are 1, then either x was inserted, or other insertions happened to set those same bits, which creates a false positive.

1.2.2 Worked Example

Consider a Bloom filter with $m = 10$ bits and $k = 3$ hash functions h_1, h_2, h_3. We begin with an empty bit array:

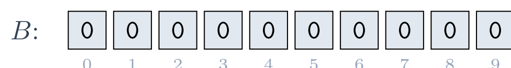


Figure 1.1. Initial state: all bits set to 0.

Insert "hello". Suppose our hash functions produce:

$$h_1(\texttt{"hello"}) = 1, \quad h_2(\texttt{"hello"}) = 4, \quad h_3(\texttt{"hello"}) = 9$$

We set $B[1], B[4], B[9] \leftarrow 1$:

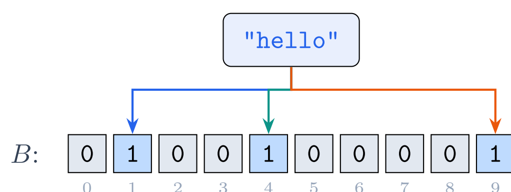


Figure 1.2. After inserting `"hello"`: bits 1, 4, and 9 are set.

Insert "world". Suppose:

$$h_1(\texttt{"world"}) = 3, \quad h_2(\texttt{"world"}) = 4, \quad h_3(\texttt{"world"}) = 7$$

We set $B[3], B[4], B[7] \leftarrow 1$. Note that $B[4]$ was already 1 from inserting `"hello"`, it remains 1. This overlap is normal and is the source of false positives.

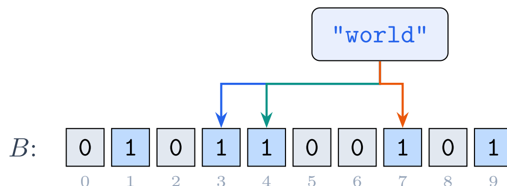


Figure 1.3. After inserting `"world"`: bits 3, 4, and 7 are set. Bit 4 was already set by `"hello"`.

Now we query three elements that were *never* inserted to illustrate the filter's behavior. **Query "cat", true negative.** Suppose:

$$h_1(\texttt{"cat"}) = 2, \quad h_2(\texttt{"cat"}) = 5, \quad h_3(\texttt{"cat"}) = 7$$

We check $B[2] = 0$. Since at least one bit is 0, the filter returns NO — `"cat"` is **definitely not** in the set. This answer is guaranteed correct.

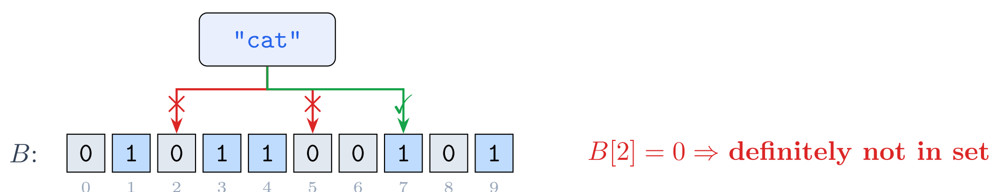


Figure 1.4. Querying `"cat"`: bit 2 is 0, so the filter correctly returns NO.

Query "dog" yields a false positive. Suppose:

$$h_1(\texttt{"dog"}) = 1, \quad h_2(\texttt{"dog"}) = 3, \quad h_3(\texttt{"dog"}) = 7$$

We check $B[1] = 1$, $B[3] = 1$, $B[7] = 1$. All three bits are set, so the filter returns YES. But `"dog"` was never inserted, bit 1 was set by `"hello"`, bit 3 by `"world"`, and bit 7 by `"world"`. This is a **false positive**: the filter is wrong, but has no way to know it.

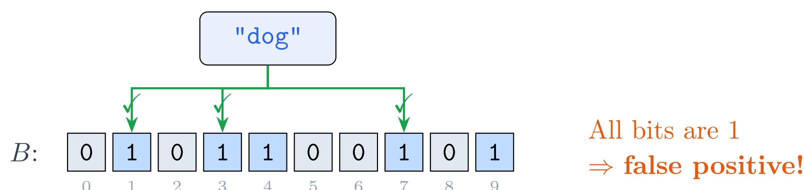


Figure 1.5. Querying `"dog"`: all checked bits happen to be 1 (set by other insertions), so the filter incorrectly returns YES. This is a false positive.

Why False Positives Happen

A Bloom filter only stores bits, not the elements themselves. When multiple elements set overlapping bits, the filter cannot distinguish whether a particular combination of 1-bits was caused by the queried element or by a coincidence of other insertions. This is the fundamental tradeoff: we use far less memory than storing elements explicitly, but we accept a small probability ϵ of false positives.

This example also reveals two important limitations of the Bloom filter that motivate the data structures in later chapters:

1. **No deletion:** We cannot remove `"hello"` from the filter by setting bits 1, 4, and 9 back to 0, as the bit 4 is shared with `"world"`, so clearing it would create a *false negative*, which Bloom filters must never produce.
2. **No expansion:** The bit array size m is fixed at creation. As more elements are inserted, the array fills with 1-bits, and the false positive rate increases. The only remedy is to rebuild the entire filter with a larger m, which requires access to all original elements.

1.2.3 False Positive Rate

We can quantify the probability of a false positive, using an analysis that follows the treatment in [4].

$$\Pr[\text{bit is picked}] = \frac{1}{m}$$

$$\Pr[\text{bit is not picked}] = 1 - \frac{1}{m}$$

As inserting on element runs k hash functions, the probability that a specific bit is not set by one element is:

$$\Pr[\text{bit not set by one element}] = \left(1 - \frac{1}{m}\right)^k$$

After inserting n elements into a Bloom filter with m bits and k hash functions, the probability that a single bit is still 0 is:

$$\Pr[\text{bit} = 0] = \left(1 - \frac{1}{m}\right)^{kn} = \left(\left(1 - \frac{1}{m}\right)^m\right)^{kn/m}$$

When m is large, we can use the approximation $(1 - 1/m)^m \approx e^{-1}$ to simplify the expression.

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

A false positive occurs when all k bits checked for a non-member happen to be 1. Since each bit is 1 with probability $1 - e^{-kn/m}$, the false positive rate is:

$$\epsilon \approx \left(1 - e^{-kn/m}\right)^k \tag{1.1}$$

To find the optimal number of hash functions k that minimizes ϵ for a given m/n, we can take the derivative of ϵ with respect to k and set it to zero.

Let $p = e^{-kn/m}$. Then ϵ can be rewritten as:

$$\epsilon = (1 - p)^k$$

Taking the natural logarithm:

$$\ln \epsilon = k \ln(1 - p)$$

Now, we differentiate with respect to k:

$$\frac{d}{dk} \ln \epsilon = \ln(1 - p) + k \cdot \frac{-1}{1 - p} \cdot \frac{dp}{dk}$$

Calculating $\frac{dp}{dk}$:

$$\frac{dp}{dk} = \frac{d}{dk} e^{-kn/m} = -\frac{n}{m} e^{-kn/m} = -\frac{n}{m} p$$

Substituting back:

$$\frac{d}{dk} \ln \epsilon = \ln(1-p) + k \cdot \frac{p}{1-p} \cdot \frac{n}{m}$$

Setting this derivative to zero for optimality:

$$\ln(1-p) + k \cdot \frac{p}{1-p} \cdot \frac{n}{m} = 0$$

Rearranging gives:

$$k \cdot \frac{p}{1-p} \cdot \frac{n}{m} = -\ln(1-p)$$

$$k = -\frac{m}{n} \ln(1-p) \cdot \frac{(1-p)}{p}$$

From our definition of p we get that:

$$k = \frac{-m \ln(p)}{n}$$

Setting the two equation equal to each other

$$-\frac{m}{n} \ln(1-p) \cdot \frac{(1-p)}{p} = -\frac{m}{n} \cdot \ln(p)$$

Cancel like terms and simplify:

$$p \ln(p) = (1-p) \ln(1-p)$$

We get that $p = 1/2$, meaning that

$$k = \frac{-m \ln(1/2)}{n} = \frac{m}{n} \ln(2)$$

Formally we can state that, for a given ratio of bits per element m/n, where m is the size of the bit array and n is the number of elements to store, the optimal number of hash functions that minimizes ϵ is:

$$k_{\text{opt}} = \frac{m}{n} \ln 2 \tag{1.2}$$

Substituting k_{opt} back into Equation 1.1 gives:

$$\epsilon_{\text{opt}} \approx \left(\frac{1}{2}\right)^{k_{\text{opt}}} = \left(\frac{1}{2}\right)^{\frac{m}{n} \ln 2} = \left(\left(\frac{1}{2}\right)^{\ln 2}\right)^{\frac{m}{n}} = (0.6185)^{\frac{m}{n}} \tag{1.3}$$

This tells us that the false positive rate decreases *exponentially* as we allocate more bits per element. The following table and figure illustrate this relationship:

Table 1.1. False positive rate vs. bits per element (with optimal k)

Bits per element (m/n)	**Optimal** k	**FPR** (ϵ)
4	3	14.7%
6	4	5.6%
8	6	2.2%
10	7	0.82%
12	8	0.31%
16	11	0.046%
20	14	0.0067%

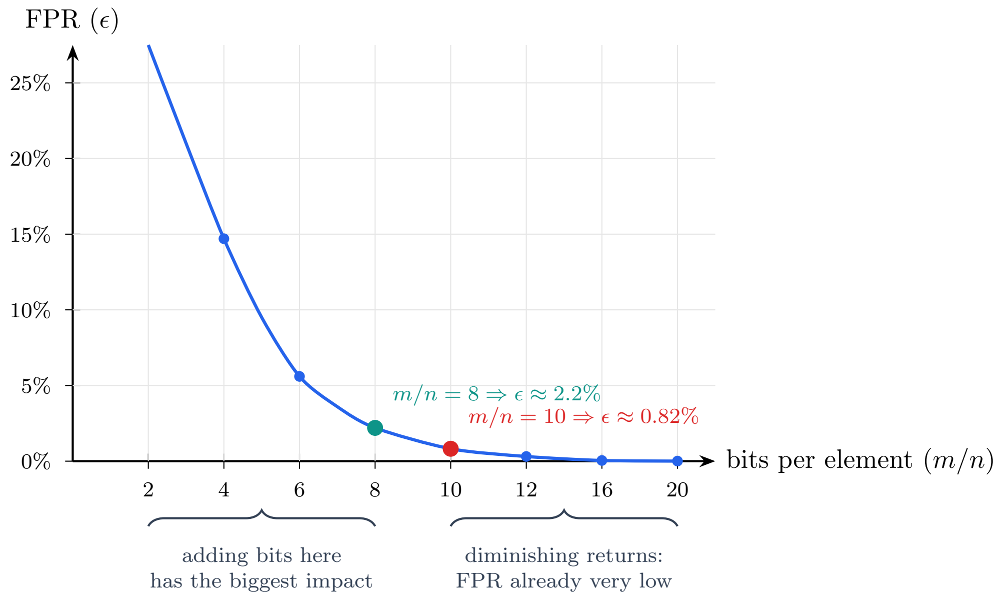


Figure 1.6. False positive rate as a function of bits per element with optimal k. The rate decreases exponentially: doubling the bits per element roughly squares the improvement. This means that as the total ratio of bits per element increases, the FPR goes down.

Equation 1.3 and Figure 1.6 reveal an important practical insight: a Bloom filter with just 10 bits per element achieves a false positive rate under 1%, regardless of how large the universe U is. This extreme space efficiency, requiring only ≈ 1.25 bytes per element for sub-1% error, is what makes Bloom filters so widely deployed.

1.2.4 Space Efficiency

From an information-theoretic perspective, any data structure that answers membership queries with a false positive rate of ϵ must use at least $\log_2(1/\epsilon)$ bits per element, this is the minimum amount of information needed to distinguish "in the set" from "not in the set" with error probability ϵ.

From Equation 1.3, we can solve for the number of bits per element required by a Bloom filter. Taking the logarithm of both sides gives:

$$\frac{m}{n} = \frac{\log_2(1/\epsilon)}{\ln 2} = 1.44 \log_2(1/\epsilon)$$

A space-optimized Bloom filter therefore uses $1.44 \log_2(1/\epsilon)$ bits per element [2, 7], which is a 44% overhead over the information-theoretic lower bound of $\log_2(1/\epsilon)$ bits. In practice, this overhead is a modest price for the simplicity and speed of the data structure.

1.3 Cuckoo Filters

Cuckoo hashing is a collision resolution scheme that guarantees worst-case constant-time lookups. While several variants exist, we present the standard formulation using two hash tables.

A cuckoo hash table consists of two tables T_1 and T_2, each of size r, along with two independent hash functions $h_1, h_2 : U \to \{0, 1, \ldots, r-1\}$. Each element $x \in S$ is stored in exactly one of two possible locations: either $T_1[h_1(x)]$ or $T_2[h_2(x)]$.

1.3.1 Structure and Operations

The data structure is initalized to

$$T_1[j] \leftarrow \varnothing \quad \text{and} \quad T_2[j] \leftarrow \varnothing \quad \forall\, j \in \{0, 1, \ldots, rr - 1\}$$

The data structure supports three operations:

1. **Search:** To determine whether $x \in S$, we examine $T_1[h_1(x)]$ and $T_2[h_2(x)]$. Since each lookup is a direct array access, search runs in $O(1)$ worst-case time.
2. **Delete:** To remove an element x, we check both candidate positions $T_1[h_1(x)]$ and $T_2[h_2(x)]$. If x occupies either location, we remove it. This operation also runs in $O(1)$ worst-case time.
3. **Insert:** To insert an element x, we attempt to place it in $T_1[h_1(x)]$. If this position is occupied by some element y, we evict y and attempt to place y in its alternate location. This process continues until either an empty slot is found or a cycle is detected, necessitating a rehash with new hash functions.

1.3.2 Worked Example

We begin with two empty tables of size 4:

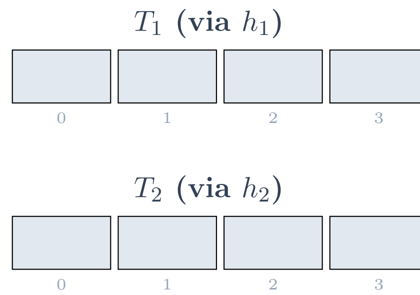


Figure 1.7. Two empty hash tables for cuckoo hashing.

Insert "A": Suppose $h_1(\texttt{A}) = 1$ and $h_2(\texttt{A}) = 2$. Slot 1 in T_1 is empty, so we place `A` there:

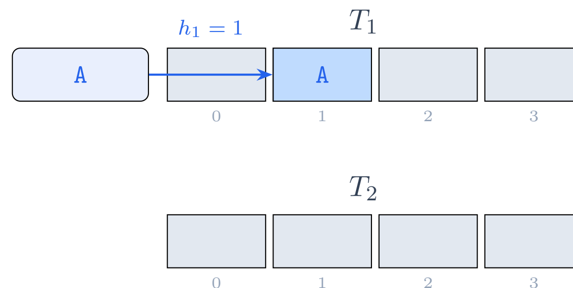


Figure 1.8. Insert `A`: placed in $T_1[1]$.

Insert "B": Suppose $h_1(\texttt{B}) = 3$ and $h_2(\texttt{B}) = 0$. Slot 3 in T_1 is empty, so we place `B` there:

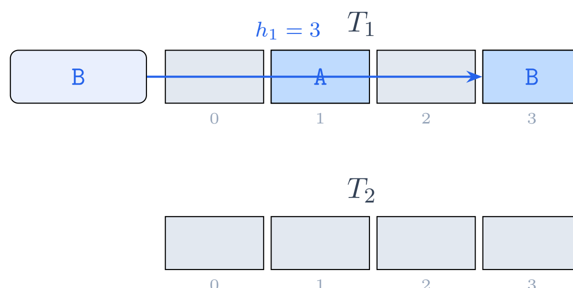


Figure 1.9. Insert `B`: placed in $T_1[3]$.

Insert "C": Suppose $h_1(\texttt{C}) = 1$ and $h_2(\texttt{C}) = 3$. Slot 1 in T_1 is occupied by `A`. So `C` *evicts* `A` from $T_1[1]$, and `A` moves to its alternative location $h_2(\texttt{A}) = 2$ in T_2:

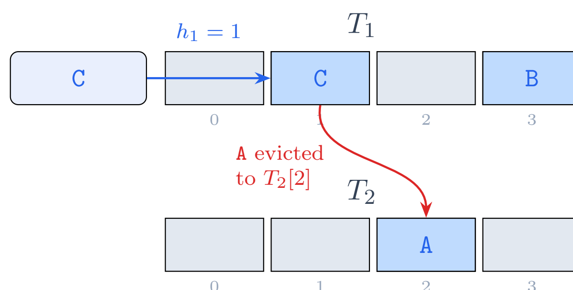


Figure 1.10. Insert `C`: evicts `A` from $T_1[1]$. `A` moves to its alternative slot $T_2[2]$.

Query "A": To look up a key, we check both possible locations: $T_1[h_1(\texttt{A})] = T_1[1]$ contains `C` (not a match), but $T_2[h_2(\texttt{A})] = T_2[2]$ contains `A`, meaning it was found. This always takes at most 2 lookups, giving $\mathcal{O}(1)$ query time.

Delete "A": To delete, we simply clear $T_2[2]$. Unlike Bloom filters, deletion is straightforward because each key occupies exactly one slot, clearing it cannot affect other keys.

Bloom Filters vs. Cuckoo Hashing

In a Bloom filter, multiple keys share the same bits, making deletion impossible without risking false negatives. In cuckoo hashing, each key has its own dedicated slot, so deletion is safe. However, the table has a fixed capacity, meaning that once both tables are sufficiently full, insertions fail and the entire structure must be rebuilt. This inability to *expand* efficiently is the limitation that quotient filters and ultimately the Aleph Filter address.

1.3.3 From Cuckoo Hashing to Cuckoo Filters

The cuckoo hashing scheme presented above stores complete keys in each table entry. A **cuckoo filter** [7] modifies this design by storing only an f-bit **fingerprint** $\phi(x)$ for each element x, where $\phi : U \to \{0,1\}^f$ is a hash function mapping elements to compact bit strings. This transformation converts an exact hash table into a probabilistic filter: it uses significantly less space, but introduces false positives when two distinct elements $x \neq y$ happen to satisfy $\phi(x) = \phi(y)$.

Fingerprint

A **fingerprint** $\phi(x)$ is an f-bit hash of element x. The parameter f controls the tradeoff between space and accuracy: larger f means more possible fingerprint values (2^f), reducing the chance of collisions but increasing memory usage.

Single-Array Structure and Bucket Positions

While our earlier presentation of cuckoo hashing used two separate tables T_1 and T_2, the cuckoo filter consolidates them into a single array B of m buckets using **partial-key cuckoo hashing** [7]. To understand how the two candidate buckets are computed, consider inserting the element `"hello"`:

1. **Hash the full key:** Compute a hash of the original element, producing a large integer:
$$H(\texttt{"hello"}) = \texttt{0xA3F7B21E9C4D8256} \quad \text{(64 bits)}$$
2. **Split the hash:** Divide the output into two non-overlapping parts:
   - The **long part** determines the first bucket index.
   - The **short part** becomes the fingerprint $\phi(x)$, the only piece stored in the table.

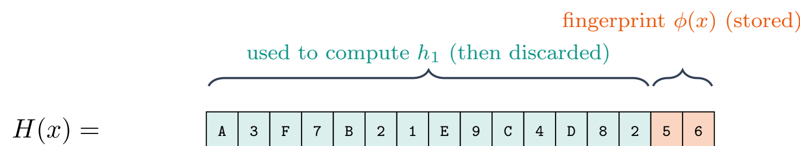


Figure 1.11. A single hash of the key is split: the long part computes the bucket index, the short part becomes the stored fingerprint. The two parts must not overlap to avoid correlation.

3. **Compute the first bucket h_1:** Apply the modulo operation to the long part. The **modulo** operation $a \mod m$ returns the remainder after dividing a by m, guaranteeing the result falls in $\{0, 1, \ldots, m-1\}$ regardless of how large the hash value is:
$$h_1(x) = \texttt{0xA3F7B21E9C4D82} \mod m$$
For example, with $m = 8$ buckets:
$$\texttt{0xA3F7B21E9C4D82} \mod 8 = 2$$
So $h_1(x) = 2$.
4. **Compute the second bucket h_2:** Hash the fingerprint with a separate hash function, then XOR the result with h_1. Suppose $G(\texttt{0x56}) \mod m = 4$:
$$h_2(x) = h_1(x) \oplus G(\phi(x)) = 2 \oplus 4 = 6$$

In general, for any element x:

$$h_1(x) = H(x) \mod m \tag{1.4}$$
$$h_2(x) = h_1(x) \oplus G(\phi(x)) \tag{1.5}$$

After insertion, only the fingerprint (`0x56`) is stored in one of the two candidate buckets. The original key `"hello"` and the long part of the hash are discarded permanently.

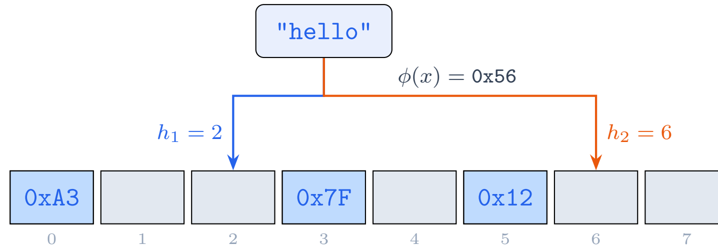


Figure 1.12. The cuckoo filter stores fingerprints in a single array. Element "hello" maps to buckets 2 and 6; only the fingerprint 0x56 is stored in one of them.

Why Re-Hash the Fingerprint?

In step 4, the fingerprint is hashed before XOR-ing with h_1. Without this, $h_2 = h_1 \oplus \phi(x)$ directly would constrain the alternate bucket to within 2^f positions of h_1 (e.g., within 256 positions for $f = 8$). Hashing the fingerprint first spreads the alternate buckets uniformly across the entire table [7].

XOR Symmetry and Eviction

The choice of XOR in Equation 1.5 is not arbitrary. XOR has a unique algebraic property: **it is its own inverse**. For any values a and b:

$$a \oplus b = c \quad \Longrightarrow \quad c \oplus b = a \tag{1.6}$$

At the bit level, XOR flips specific bits. Applying it again flips them back, restoring the original value:

$$\underset{2}{\texttt{010}} \oplus \underset{4}{\texttt{100}} = \underset{6}{\texttt{110}} \qquad \text{and} \qquad \underset{6}{\texttt{110}} \oplus \underset{4}{\texttt{100}} = \underset{2}{\texttt{010}}$$

This means that given *either* bucket index and the hashed fingerprint, the *other* bucket is always recoverable:

$$\underbrace{h_1(x)}_{2} \oplus \underbrace{\text{hash}(\phi(x))}_{4} = \underbrace{h_2(x)}_{6} \qquad \text{and} \qquad \underbrace{h_2(x)}_{6} \oplus \underbrace{\text{hash}(\phi(x))}_{4} = \underbrace{h_1(x)}_{2}$$

During eviction, the filter holds a fingerprint in some bucket i but does not know whether i is h_1 or h_2, it does not matter. The same formula always yields the alternate bucket:

$$j = i \oplus \text{hash}(\phi(x)) \tag{1.7}$$

This is essential because the original key x has been discarded. Without XOR symmetry, the filter would need to store the full key to recompute $h_1(x) = \text{hash}(x) \mod m$, defeating the purpose of using fingerprints.

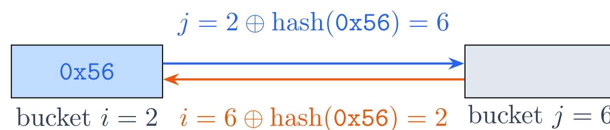


Figure 1.13. XOR symmetry enables eviction without the original key: given any bucket index and the fingerprint, the alternate bucket is always computable.

Bucket Structure

Each bucket holds up to b fingerprints (typically $b = 4$). An insertion requires eviction only when all $2b$ slots across both candidate buckets are occupied. With $b = 4$, the table achieves a **load factor** of $\alpha \approx 0.955$ before insertions begin to fail [7]. The load factor α is the fraction of slots currently occupied, $\alpha = 0.955$ means 95.5% of all slots in the table are filled.

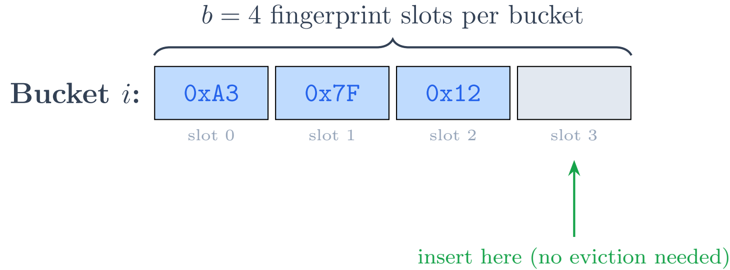


Figure 1.14. A single bucket with $b = 4$ slots. Three fingerprints are stored; one slot remains empty. An insertion into this bucket takes the empty slot without triggering eviction.

Operations

The cuckoo filter supports three operations, each in $\mathcal{O}(1)$ time:

1. **Insertion:** To insert element x, compute $\phi(x)$, $h_1(x)$, and $h_2(x)$. If either bucket has an empty slot, place $\phi(x)$ there. If both buckets are full (all $2b$ slots occupied), evict a random fingerprint from one bucket and relocate it to its alternate bucket via Equation 1.7. This process repeats until an empty slot is found or a maximum number of evictions is reached.

2. **Query:** To test whether $x \in S$, compute $\phi(x)$, $h_1(x)$, and $h_2(x)$, then check both candidate buckets:

$$\text{query}(x) = \big(\phi(x) \in B[h_1(x)]\big) \ \vee \ \big(\phi(x) \in B[h_2(x)]\big)$$

   If no matching fingerprint is found, x is **definitely not** in the set. If a match is found, x is **probably** in the set, as the match may be a false positive caused by a different element with the same fingerprint.

3. **Deletion:** To remove x, locate $\phi(x)$ in either $B[h_1(x)]$ or $B[h_2(x)]$ and remove one copy. This is safe because each fingerprint occupies its own dedicated slot, removing it cannot affect other elements. This is a key advantage over Bloom filters, where bits are shared and deletion would risk creating false negatives.

False Positive Rate

A false positive occurs when a non-member x has a fingerprint matching one already stored in a candidate bucket. A query checks two buckets of b entries each, at most $2b$ fingerprints. Each f-bit fingerprint matches with probability $1/2^f$, so:

$$\epsilon \approx \frac{2b}{2^f} \tag{1.8}$$

Solving for the minimum fingerprint size f:

$$2^f \geq \frac{2b}{\epsilon} \quad \Longrightarrow \quad f \geq \log_2(1/\epsilon) + \log_2(2b)$$

With $b = 4$, $\log_2(2 \cdot 4) = 3$. Applying the semi-sorting optimization from [7], which saves one bit per fingerprint by exploiting the irrelevance of fingerprint ordering within a bucket:

$$f = \log_2(1/\epsilon) + 2 \tag{1.9}$$

Space Efficiency

Since not every slot is occupied, the amortized cost per stored item is the fingerprint size divided by the load factor α:

$$C_{\text{cuckoo}} = \frac{f}{\alpha} = \frac{\log_2(1/\epsilon) + 2}{\alpha} \quad \text{bits per item} \tag{1.10}$$

Space Comparison at $\epsilon = 1\%$

Cuckoo filter ($b = 4$, $\alpha = 0.955$):

$$C_{\text{cuckoo}} = \frac{\log_2(100) + 2}{0.955} = \frac{6.64 + 2}{0.955} \approx 9.1 \text{ bits/item}$$

Bloom filter (Equation 1.3):

$$C_{\text{Bloom}} = 1.44 \cdot \log_2(100) = 1.44 \times 6.64 \approx 9.6 \text{ bits/item}$$

At $\epsilon = 1\%$, the cuckoo filter uses 5% less space while also supporting deletion.

Table 1.2. Space comparison: Cuckoo filter ($b = 4$, $\alpha = 0.955$) vs. Bloom filter at various false positive rates.

FPR (ϵ)	**Cuckoo** (bits/item)	**Bloom** (bits/item)	**Winner**
10%	5.57	4.78	Bloom
3%	7.39	7.29	$\approx$ tie
1%	9.05	9.57	Cuckoo
0.1%	12.53	14.35	Cuckoo
0.01%	16.01	19.13	Cuckoo

The crossover occurs at $\epsilon \approx 3\%$: below this threshold, the cuckoo filter's fixed +2 overhead (Equation 1.9) becomes a diminishing fraction of the growing $\log_2(1/\epsilon)$ term, while the Bloom filter's 1.44× multiplicative factor scales proportionally.

1.4 Summary of Filter Limitations

The cuckoo filter solves the Bloom filter's deletion problem but inherits a critical limitation: **it cannot expand**. The array size m is fixed at construction. When the load factor approaches α, insertions fail. Expanding to a larger table of size $m' > m$ would require recomputing:

$$h_1(x) = \text{hash}(x) \mod m'$$

for every stored element. But the original keys have been discarded, only fingerprints remain, and $\phi(x)$ does not contain enough information to reconstruct $\text{hash}(x)$.

The Expansion Problem

Both Bloom filters and cuckoo filters have fixed capacity. The Bloom filter cannot delete; the cuckoo filter cannot expand. A filter that supports *both* deletion and efficient expansion requires a fundamentally different approach to storing fingerprints, one where expansion does not require access to the original keys. This is the problem that quotient filters address, and that the Aleph Filter ultimately solves with $\mathcal{O}(1)$ operations.

Table 1.3. Capabilities and limitations of the filters discussed so far. The gaps in this table motivate the quotient filter family and the Aleph Filter.

Property	Bloom	Cuckoo	Needed
Insert	$\mathcal{O}(k)$	$\mathcal{O}(1)^*$	$\mathcal{O}(1)$
Query	$\mathcal{O}(k)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Delete	×	✓	✓
Expand	×	×	✓
Stable FPR	×	✓	✓

* = amortized. k = number of hash functions.

The Core Limitation

Bloom filters can't delete. Cuckoo filters can't expand efficiently. Quotient filters *can* expand, but their query time degrades to $\mathcal{O}(\log N)$ as void entries accumulate. The Aleph Filter solves all three.

Chapter 2

Quotient Filters and Expansion

The previous chapter established that Bloom filters cannot delete and cuckoo filters cannot expand. Both limitations stem from the same root cause: the stored representation (bits or fingerprints) does not retain enough information to reconstruct the element's position in a resized table.

Quotient filters, introduced by Bender et al. [8], solve the expansion problem with an elegant insight: instead of computing the bucket index and fingerprint from *separate* parts of the hash (as cuckoo filters do), they derive both from a *single* hash by splitting it at a chosen bit position. This means the bucket index and fingerprint together always reconstruct the full hash, and expansion simply moves the split point.

2.1 The Quotienting Idea

Quotienting

Given a p-bit hash $h(x)$, we split it into two parts at bit position q:

- **Quotient**: the top q bits, denoted $h_q(x) \to$ determines the *slot address*
- **Remainder**: the bottom $r = p - q$ bits, denoted $h_r(x) \to$ stored as the *fingerprint*

The table has 2^q slots. Together, the quotient and remainder reconstruct the full hash: $h(x) = h_q(x) \| h_r(x)$, where $\|$ denotes concatenation.

This is fundamentally different from the cuckoo filter's approach. Recall that in a cuckoo filter, the bucket index comes from $H(x) \mod m$ and the fingerprint comes from a separate portion of the hash. The modulo operation is **lossy**: multiple distinct hash values map to the same remainder (e.g., both $10 \mod 8 = 2$ and $18 \mod 8 = 2$), so knowing $H(x) \mod m$ does not reveal what $H(x) \mod m'$ would be for a different table size m'. This lost information is precisely what prevents cuckoo filters from expanding. In a quotient filter, the concatenating them reconstructs the original value exactly, so no information is ever lost.

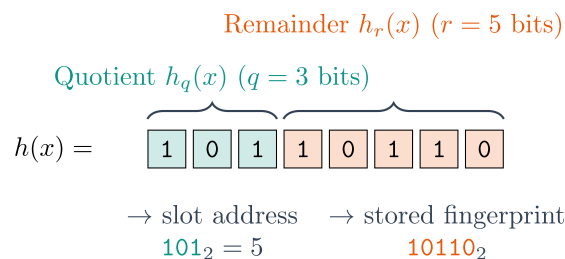


Figure 2.1. An 8-bit hash split into a 3-bit quotient (slot address) and 5-bit remainder (stored fingerprint). The table has $2^3 = 8$ slots.

Cuckoo vs. Quotient: The Key Difference

In a **cuckoo filter**, the bucket index and fingerprint are *independent*, which means knowing one tells you nothing about the other. In a **quotient filter**, they are *complementary*, and together they reconstruct the full hash. This complementarity is what enables expansion without access to the original keys.

2.1.1 Worked Example: Insertion

Consider a quotient filter with $p = 8$-bit hashes, $q = 3$-bit quotients, and $r = 5$-bit remainders. The table has $2^3 = 8$ slots. Suppose we insert three elements:

$$h(\texttt{"hello"}) = \underbrace{101}_{\text{quotient}}\ \underbrace{10101}_{\text{remainder}} \quad \rightarrow \text{slot } 5, \text{ store } \texttt{10101}$$

$$h(\texttt{"world"}) = \underbrace{011}_{\text{quotient}}\ \underbrace{00110}_{\text{remainder}} \quad \rightarrow \text{slot } 3, \text{ store } \texttt{00110}$$

$$h(\texttt{"cat"}) = \underbrace{101}_{\text{quotient}}\ \underbrace{11010}_{\text{remainder}} \quad \rightarrow \text{slot } 5, \text{ store } \texttt{11010}$$

The first two elements go directly into their home slots. But `"cat"` also hashes to slot 5, which is already occupied by `"hello"`. This is a **soft collision**: different keys with the same quotient but different remainders. The quotient filter resolves this by storing `"cat"`'s remainder in the next available slot (slot 6) using linear probing. Remainders within a run are kept in sorted order, so `"hello"`'s remainder (10101) stays in the home slot and `"cat"`'s larger remainder (11010) is placed after it:

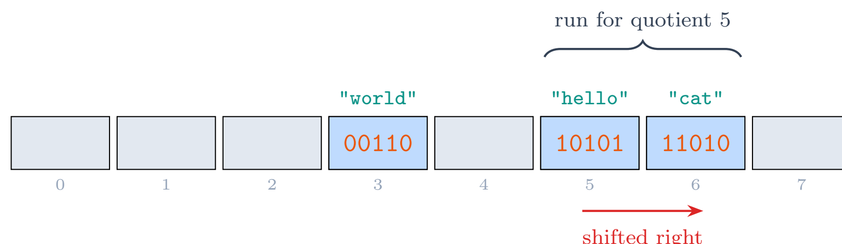


Figure 2.2. After inserting `"hello"`, `"world"`, and `"cat"`. Elements with the same quotient form a **run** stored in contiguous slots.

The set of remainders sharing the same quotient is called a **run**. A sequence of contiguous runs is called a **cluster**. As the table fills, clusters grow longer, which increases lookup time, a limitation we will revisit.

2.2 Metadata Bits

Storing remainders in shifted positions creates an ambiguity: given a remainder in some slot, how do we know which quotient it belongs to? The quotient filter resolves this with three metadata bits per slot [8]:

Metadata Bits

Each slot j in the quotient filter maintains three single-bit flags:

1. `is_occupied`: Set to 1 if quotient j is the home slot for *some* element in the filter (the element may be stored elsewhere due to shifting).
2. `is_continuation`: Set to 1 if this slot holds a remainder that is *not the first* in its run (i.e., it continues a run from a previous slot).
3. `is_shifted`: Set to 1 if the remainder in this slot has been displaced from its home slot (i.e., it does not occupy the slot matching its quotient).

These three bits encode enough information to reconstruct which quotient each remainder belongs to, without storing the quotient explicitly. The key relationships are:

- A slot with `is_occupied = 0`, `is_continuation = 0`, `is_shifted = 0` is **empty**.
- A slot with `is_shifted = 0` and `is_continuation = 0` marks the **start of a cluster**, meaning its the first element of a run in its home slot.
- `is_continuation = 0` marks the **start of a new run** within a cluster.
- `is_occupied` is attached to the *slot index*, not the stored remainder. It says "some element with this quotient exists somewhere in the table."

2.2.1 Worked Example: Metadata in Action

Continuing from Figure 2.2, suppose we also insert an element with quotient 6:

$$h(\texttt{"dog"}) = \underbrace{110}_{\text{quotient}}\ \underbrace{01001}_{\text{remainder}} \quad \rightarrow \text{slot } 6$$

Slot 6 is occupied by `"cat"` (which was shifted from slot 5). So `"dog"`'s remainder gets pushed to slot 7. The full state:

run for $q = 5$ (slots 5–6)

Slot	3	4	5	6	7
Remainder	00110		10101	11010	01001
`is_occupied`	1	0	1	1	0
`is_cont.`	0	0	0	1	0
`is_shifted`	0	0	0	1	1
Belongs to	$q = 3$	—	$q = 5$	$q = 5$	$q = 6$

cluster (contiguous occupied region) (slots 5–7)

Figure 2.3. Quotient filter state after inserting `"world"` ($q = 3$), `"hello"` ($q = 5$), `"cat"` ($q = 5$), and `"dog"` ($q = 6$). The metadata bits encode run boundaries and shift status.

Reading the metadata for each slot:

- **Slot 3**: `is_occupied = 1` (quotient 3 has elements), `is_shifted = 0` (it is in its home slot), `is_continuation = 0` (first in its run). This is `"world"`'s home.

- **Slot 5**: `is_occupied = 1` (quotient 5 has elements), `is_shifted = 0` (in its home slot), `is_continuation = 0` (starts a new run). This is `"hello"`.
- **Slot 6**: `is_occupied = 1` (quotient 6 has elements somewhere), `is_shifted = 1` (not in its home slot), `is_continuation = 1` (continues the run from slot 5). This is `"cat"`, which belongs to quotient 5 but is stored in slot 6.
- **Slot 7**: `is_occupied = 0` (no element has quotient 7), `is_shifted = 1` (displaced from slot 6), `is_continuation = 0` (starts a new run). This is `"dog"`, which belongs to quotient 6.

2.2.2 Lookup Algorithm

To query whether an element x with quotient q and remainder r is in the filter:

1. Check slot q. If `is_occupied = 0`, then no element with quotient q exists, return NO.
2. **Find the cluster start**: scan left from slot q until a slot with `is_shifted = 0` is found. This is where the cluster begins.
3. **Skip preceding runs**: scan right through the cluster, counting runs. Each `is_occupied = 1` slot to the left of q indicates a run to skip. Each `is_continuation = 0` slot marks the start of a new run. When the count reaches zero, the current position is the start of quotient q's run.
4. **Search the run**: compare the remainder in each slot of the run to r. If a match is found, return YES (probably in the set). If the run ends without a match, return NO (definitely not in the set).

Lookup for `"dog"` (quotient $= 6$)

1. Slot 6: `is_occupied = 1` $\rightarrow$ quotient 6 has elements. Proceed.
2. Scan left: slot 6 has `is_shifted = 1`, slot 5 has `is_shifted = 0` $\rightarrow$ cluster starts at slot 5.
3. Scan right from slot 5, counting runs. Between slot 5 and slot 6, there is one `is_occupied` slot (slot 5) before our target (slot 6), so we skip one run:
   - Slot 5: `is_continuation = 0` $\rightarrow$ start of run for $q = 5$ (skip this run)
   - Slot 6: `is_continuation = 1` $\rightarrow$ still $q = 5$'s run
   - Slot 7: `is_continuation = 0` $\rightarrow$ new run starts $\rightarrow$ this is $q = 6$'s run!
4. Slot 7 contains remainder `01001`. Compare to `"dog"`'s remainder `01001` $\rightarrow$ match! Return YES.

Notice that this lookup required scanning through multiple slots. In the worst case, if all elements hash to the same quotient, a single run could span the entire table, requiring $O(n)$ time. In practice with a good hash function, runs are short and expected lookup time is $O(1)$. However, as the load factor increases, clusters grow and performance degrades, quotient filters are typically not filled beyond 75% [8].

2.3 How Expansion Works

This is where the quotient filter's design pays off. Because the quotient and remainder together form the complete hash, expanding the table requires no access to the original keys, only a reinterpretation of the stored information.

2.3.1 The Expansion Procedure

To double the table from 2^q to 2^{q+1} slots:

1. **Move the split point**: the quotient grows from q bits to $q+1$ bits, and the remainder shrinks from r bits to $r-1$ bits.
2. **Redistribute entries**: for each stored remainder, the topmost bit of the old remainder becomes part of the new quotient. If this bit is 0, the entry stays in its current slot. If it is 1, the entry moves to the corresponding slot in the new (upper) half of the table.
3. **Truncate remainders**: remove the top bit from each remainder, since it is now part of the quotient.

Expansion Step by Step

Before expansion ($q=3$, $r=5$, table has $2^3=8$ slots):

$$h(\texttt{"hello"}) = \underbrace{101}_{\text{quotient}}\ \underbrace{10101}_{\text{remainder}} \quad \rightarrow \text{slot } 5, \text{ store } \texttt{10101}$$

After expansion ($q=4$, $r=4$, table has $2^4=16$ slots):

$$h(\texttt{"hello"}) = \underbrace{1011}_{\text{new quotient}}\ \underbrace{0101}_{\text{new remainder}} \quad \rightarrow \text{slot } 11, \text{ store } \texttt{0101}$$

The hash `10110101` did not change. We simply read one more bit as part of the quotient. The top bit of the old remainder (`1`) became the new lowest bit of the quotient, changing the slot from $\texttt{101}_2 = 5$ to $\texttt{1011}_2 = 11$.

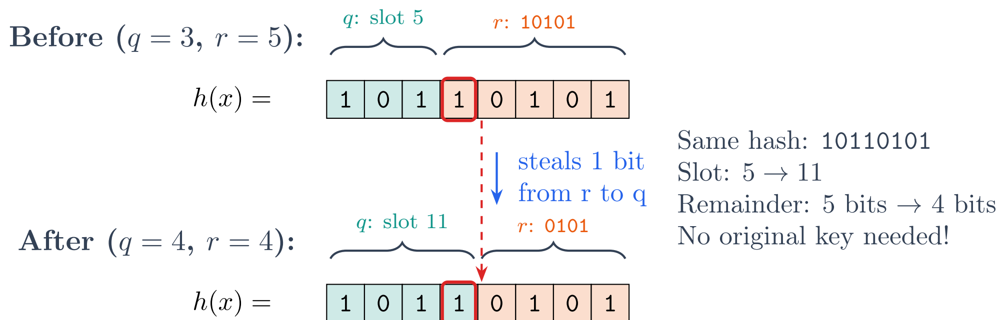
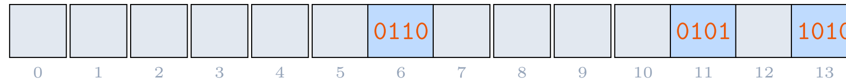


Figure 2.4. Expansion moves the split point one bit to the right. The top bit of the old remainder (highlighted) becomes the new lowest bit of the quotient. The hash itself is unchanged.

Before Expansion (8 slots)

expand: steal 1 bit from each remainder

After Expansion (16 slots)



Remainders shrink: 5 bits → 4 bits. Entries redistribute across doubled table.

Figure 2.5. Quotient filter expansion: the table doubles from 8 to 16 slots. Each remainder loses its top bit to the quotient. `"hello"` moves from slot 5 to slot 11; `"cat"` moves from slot 6 to slot 13.

Why This Works Without Original Keys

The expansion procedure never needs the original element x or the full hash hash(x). It only needs the *quotient* (known from the slot position) and the *remainder* (stored in the slot). Together these reconstruct the full p-bit hash, which can then be re-split at the new position. This is the fundamental advantage of quotienting over the cuckoo filter's modulo-based addressing.

2.4 The Void Entry Problem

Expansion comes at a cost: each time the table doubles, every remainder loses one bit. For an entry that was present at the very first expansion and survives through X expansions, its remainder shrinks from r bits to $r - X$ bits.

2.4.1 How Void Entries Arise

Consider an entry inserted with an initial remainder of $r = 5$ bits. After each expansion, the top bit of its remainder is consumed by the growing quotient:

Expansion	Remainder	Bits left
0 (initial)	1 0 1 0 1	5
1	0 1 0 1	4
2	1 0 1	3
3	0 1	2
4	1	1
5	? ? ? ? ?	**0** = VOID

No fingerprint bits left! Matches **any** query → 100% false positive

Figure 2.6. A remainder shrinks by one bit per expansion. After r expansions, zero bits remain, the entry becomes VOID.

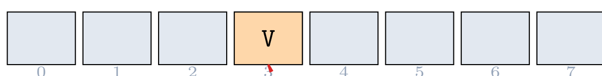
Void Entry

A **void entry** is an entry whose remainder has been reduced to zero bits through repeated expansions. Since there is no fingerprint left to compare against, a void entry matches *every* query to its slot, producing a guaranteed false positive. Formally, if an entry was created with remainder length r and has survived $X \geq r$ expansions, its current remainder length is $\max(0, r - X) = 0$.

2.4.2 The Expansion Dilemma

Void entries create a specific problem during expansion. When the table doubles, each entry moves to one of two possible slots in the new table, determined by the top bit of its current remainder. But a void entry has *no remaining bits*. Without a bit to read, the filter cannot determine which of the two new slots the entry belongs in:

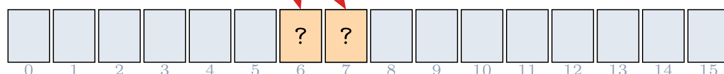
Before expansion (8 slots)



bit = 0?
bit = 1?

No remainder bits left
to decide which slot!

Slot 6 (if bit was 0)
or slot 7 (if bit was 1)
We don't know.



After expansion (16 slots)

Figure 2.7. A void entry at slot 3 must move to either slot 6 or slot 7 during expansion, but with zero remainder bits, there is no information to determine which. This is the core problem that InfiniFilter and the Aleph Filter each solve differently.

2.4.3 InfiniFilter's Approach: Secondary Tables

InfiniFilter [6] solves this by placing void entries into **secondary hash tables**, one per expansion level. When an entry becomes void, it is moved to a secondary table that stores enough additional hash bits to track its position.

However, this means that queries for non-existent or old keys must now search the main table *and* all secondary tables:

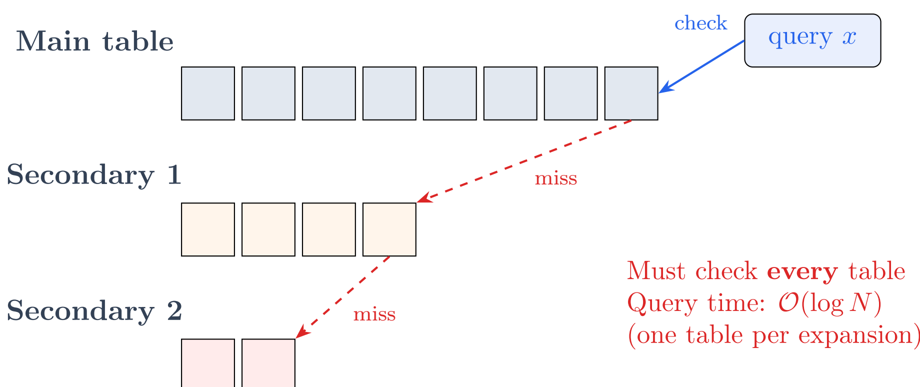


Figure 2.8. InfiniFilter stores void entries in secondary hash tables. Queries must search all tables, degrading to $\mathcal{O}(\log N)$.

With each expansion adding a new secondary table, the number of tables grows as $\mathcal{O}(\log N)$ (where N is the total number of insertions). A query that targets a non-existent key or an old key must search every table, giving worst-case $\mathcal{O}(\log N)$ query time, a significant degradation from the $\mathcal{O}(1)$ queries of the original quotient filter.

2.4.4 The Aleph Filter's Solution: Void Duplication

The Aleph Filter [5] takes a fundamentally different approach. Instead of moving void entries to secondary tables, it **duplicates** each void entry into *both* possible slots during expansion. The two candidate slots in the expanded table are $2q$ and $2q+1$, corresponding to appending a 0 or 1 bit to the old quotient:

$$q\|0 = 2q \qquad \text{and} \qquad q\|1 = 2q+1$$

where $\|$ denotes bit concatenation. A comparison betwen InfiniFilter and Aleph Filter:

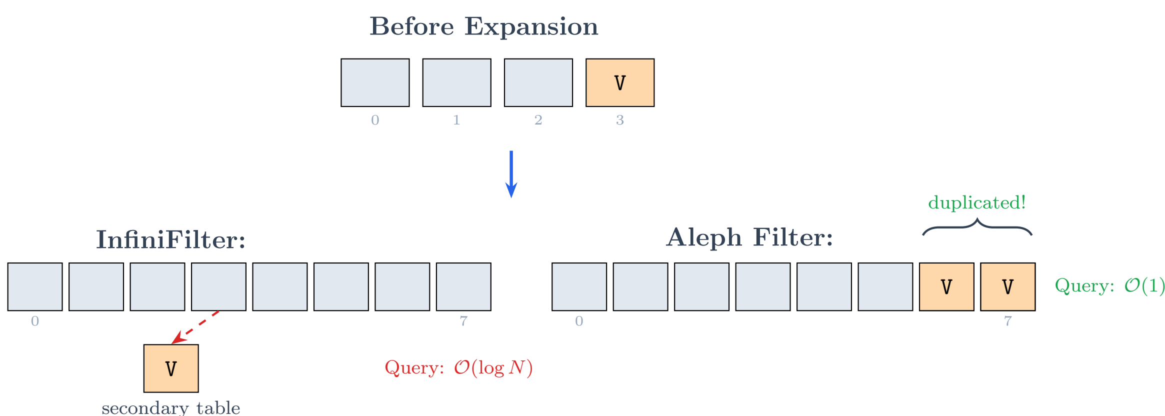


Figure 2.9. Aleph Filter duplicates void entries to both possible slots, keeping queries $\mathcal{O}(1)$.

Since a void entry matches any query to its slot (it has no fingerprint to compare), placing a copy in both candidate slots ensures correctness: regardless of which slot the original element "should" have gone to, a copy is there. All entries remain in a single table, so queries never need to search secondary structures.

This is the core innovation of the Aleph Filter, which the next chapter develops in full detail. The complete design addresses three challenges:

1. **Void duplication:** keeps queries $\mathcal{O}(1)$ by eliminating secondary tables.
2. **Tombstone-based deletion:** handles the complication that deleting a duplicated void entry must not remove both copies.
3. **Rejuvenation:** prevents void entries from accumulating indefinitely by periodically replacing them with fresh fingerprints.

The Complete Filter Evolution

	Delete	**Expand**	**Query**	**Stable FPR**
Bloom filter	$\times$	$\times$	$\mathcal{O}(k)$	$\times$
Cuckoo filter	✓	$\times$	$\mathcal{O}(1)$	$\times$
Quotient filter	✓	✓	$\mathcal{O}(1)$*	$\times$
InfiniFilter	✓	✓	$\mathcal{O}(\log N)$	✓
Aleph Filter	✓	✓	$\mathcal{O}(1)$	✓

*amortized; degrades at high load factors

Each row solves a limitation of the row above it. The Aleph Filter is the first to achieve all four properties simultaneously.

Chapter 3

The Aleph Filter

3.1 Overview

The previous chapter established that quotient filters can expand without access to the original keys by moving the quotient-remainder split point. However, after enough expansions, old entries run out of fingerprint bits and become *void entries*. InfiniFilter [6] handles void entries by moving them to secondary hash tables, but this degrades query time to $\mathcal{O}(\log N)$ as more tables accumulate.

The Aleph Filter [5] builds directly on InfiniFilter's architecture, using the same slot format, variable-length fingerprints, and unary padding, but changes how void entries are handled during expansion. This single change, combined with two supporting mechanisms, yields $\mathcal{O}(1)$ worst-case time for all operations while expanding infinitely.

This chapter presents the three contributions in order. We begin with the slot format inherited from InfiniFilter, then introduce void entry duplication (Section 3.3), the core idea that enables constant-time queries (Section 3.4). We then cover tombstone-based deletes (Section 3.5) and rejuvenation (Section 3.6), two mechanisms that keep the filter efficient under updates. Finally, we analyze the FPR (Section 3.7), describe the operating regimes (Section 3.8), and summarize the full architecture.

The Three Contributions

1. **Void entry duplication:** during expansion, void entries are copied to *both* possible new slots, keeping everything in one table $\rightarrow \mathcal{O}(1)$ queries.
2. **Tombstone-based deletes:** deleting a void entry replaces it with a tombstone and defers duplicate removal to the next expansion $\rightarrow \mathcal{O}(1)$ deletes.
3. **Rejuvenation:** void entries are periodically replaced with fresh fingerprints, preventing unbounded accumulation of duplicates $\rightarrow$ stable FPR.

3.2 Structure

The Aleph Filter inherits InfiniFilter's slot format. Each slot in the main hash table stores three metadata bits followed by a data region containing a variable-length fingerprint and a self-delimiting unary code [6, 5]:

Slot format (example: 12-bit slots = 3 metadata + 9 data bits)

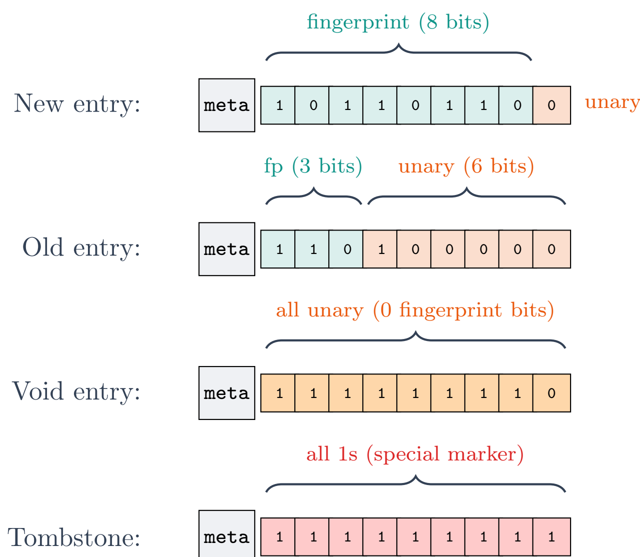


Figure 3.1. The four possible slot states in an Aleph Filter. The unary code is self-delimiting: read 1s until a 0 is encountered, then the remaining bits are the fingerprint. A void entry has no fingerprint; a tombstone is distinguished by an all-1s pattern.

The unary code allows the filter to store variable-length fingerprints within fixed-width slots. Newer entries receive longer fingerprints (more information, fewer false positives), while older entries that have survived multiple expansions have shorter fingerprints. This is the key mechanism inherited from InfiniFilter that allows infinite expansion [6].

It is important to note that only the stored fingerprint shrinks across expansions. The hash function always produces a fixed-length output (e.g., 64 bits). When a new entry is inserted into a large table, the hash is split at the current table size: more bits go to the quotient (slot address), but the fingerprint is still initialized to F bits. The void problem only affects *old* entries, because they were inserted when the table was smaller and only a small slice of the hash was saved. The bits that were discarded at insertion time cannot be recovered, so after enough expansions the stored slice is fully consumed. New entries, by contrast, always have access to the full hash output and simply reach deeper into it for the larger quotient while still reserving F bits for the fingerprint:

$$\underbrace{q\text{ bits}}_{\log_2(\text{table size})} \;\Big|\; \underbrace{F\text{ bits}}_{\text{fingerprint}} \;\Big|\; \underbrace{\text{unused bits}}_{\text{discarded}}$$

As the table grows, q increases and the unused portion shrinks, but F stays the same for every new insertion. In theory, if the table ever reached 2^{64} slots (far beyond any practical system), even new entries would have no room for fingerprints.

3.2.1 The Mother Hash

As an entry survives expansions, the quotient-remainder split point slides through its hash: fingerprint bits are consumed one by one to extend the quotient (slot address). By the time an entry becomes void, its slot address in the main table contains all the hash bits that were ever stored for it: the original quotient bits plus every fingerprint bit that was transferred across expansions. This accumulated slot address is the entry's **mother hash**.

A mother hash inserted at Generation j (when the table had 2^j slots and the quotient was j bits) and that had F fingerprint bits at insertion will have $j + F$ total bits when it becomes void. An entry inserted later, at Generation j' where $j' > j$, started with a

longer quotient (j' bits) and the same F fingerprint bits, giving a mother hash of $j' + F$ bits. Therefore, **newer entries have longer mother hashes than older entries**.

Understanding the mother hash is essential for the delete and rejuvenation mechanisms described in Sections 3.5 and 3.6. The mother hash tells the filter where all of a void entry's duplicates are located, enabling their removal.

With the slot format and mother hash concept in place, we now turn to the Aleph Filter's core contribution: how it handles void entries during expansion.

3.3 Void Entry Duplication

This section describes the Aleph Filter's core innovation. During expansion, each non-void entry moves to one of two possible slots in the expanded table. The lowest bit of its remainder determines which slot: if the old quotient is q, the new slot is either $2q$ (if the bit is 0) or $2q + 1$ (if the bit is 1). After moving, that bit is removed from the remainder.

A void entry has no remaining bits. It cannot determine which of the two new slots it belongs to. InfiniFilter solves this by moving void entries to secondary hash tables [6]. The Aleph Filter takes a different approach: **duplicate the void entry into both possible slots** [5].

The intuition is straightforward: since the filter does not know whether the missing bit would have been 0 or 1, it places the entry in both branches. On the next expansion, each copy faces the same ambiguity and is again placed in both child slots. This continues for every subsequent expansion, so no matter what the original key's full hash would have resolved to, a void entry is guaranteed to be waiting at the correct slot.

Void Duplication

During expansion, for each void entry at slot q in the old table, the Aleph Filter places a void entry at *both* slot $2q$ and slot $2q + 1$ in the expanded table. This ensures that regardless of which slot the original element's hash would have directed it to, a matching entry is present.

3.3.1 Worked Example

Consider a filter with 4 slots (2^2) containing a void entry at slot 3 (binary: `11`). During expansion to 8 slots (2^3), slot 3 could map to either slot 6 (`110`) or slot 7 (`111`):

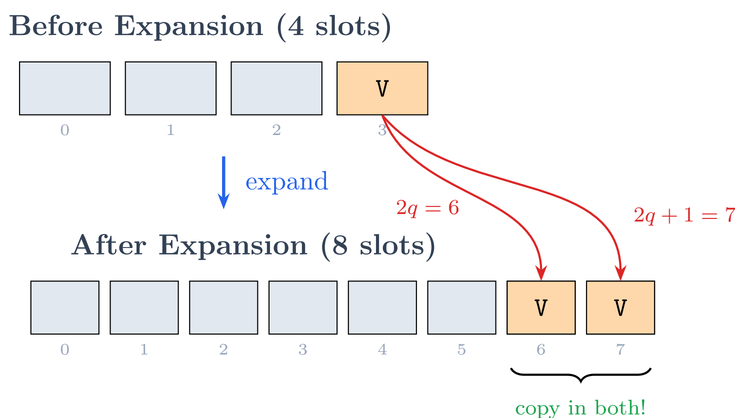


Figure 3.2. A void entry at slot 3 is duplicated to slots 6 and 7 during expansion. The filter cannot determine which is correct, so it places a copy in both.

On the next expansion (8 → 16 slots), each void copy duplicates again. The copy at slot 6 (`110`) goes to slots 12 (`1100`) and 13 (`1101`). The copy at slot 7 (`111`) goes to slots 14 (`1110`) and 15 (`1111`). The entry now has 4 copies:

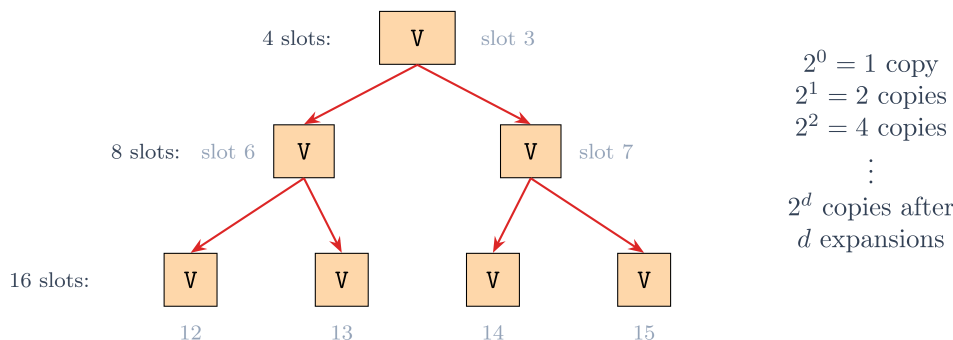


Figure 3.3. Void duplicates grow exponentially: each expansion doubles every existing copy. However, the table also doubles, so the *fraction* of void entries remains constant per generation.

3.3.2 Why Duplication Is Correct

A void entry has zero fingerprint bits. When a query encounters a void entry in its target run, the comparison is vacuously true: there are no bits to disagree on. Therefore, a void entry matches *any* query to its slot. This property ensures correctness in two ways:

1. **No false negatives:** The original element's hash determines exactly one of the two candidate slots. Since the void entry occupies *both*, it is guaranteed to be found regardless of which slot the query reaches.
2. **False positives are bounded:** A void entry in the "wrong" slot may cause additional false positives. However, since the void entry would have matched any query anyway (zero bits to compare), the extra copy does not change the per-query false positive probability. It only affects which queries encounter it.

3.3.3 How Queries Find Void Entries

It is important to understand why duplication guarantees that a query always finds the right slot despite the entry being void. At query time, the caller has access to the original key and can hash it fresh. The hash function produces a full output (e.g., 64 bits), regardless of how many bits the filter stored internally. The query computes $\log_2(\text{table size})$ bits for the quotient and goes directly to that one canonical slot. The query never needs to "search" across all the duplicates; it knows exactly where to look because it has the full hash.

The duplication exists for a different reason: during *expansion*, the filter does not have the original key. It only has the stored fingerprint. When that fingerprint is empty (void), the filter cannot decide which child slot the entry belongs to, so it places the entry in both. This ensures that whichever slot the full hash eventually resolves to at query time, a void entry will be waiting there.

3.3.4 Why Duplication Doesn't Explode

The exponential growth of void copies (Figure 3.3) appears alarming. However, two exponential effects work against each other and cancel exactly.

Older generations (entries inserted long ago) need the *most* duplicates per entry, because they have been void through many expansions. But older generations also contributed the *fewest* entries, because the table was much smaller when they were inserted. Conversely,

recently void generations contributed many entries but need very few duplicates (perhaps just one). These two effects, the exponential growth in duplicates and the exponential shrinkage in generation size, offset each other precisely.

To derive this formally, consider the table starting at size C_0. Each generation doubles the table, so Generation j sees a table of size $C_0 \cdot 2^j$ and Generation X (the current one) sees size $C_0 \cdot 2^X$. Generation j added roughly half the table's entries at the time of its expansion. The number of new entries contributed by Generation j is approximately $C_0 \cdot 2^{j-1}$ (the difference between the table size at Generation j and Generation $j-1$). The fraction of total entries that came from Generation j is therefore:

$$f(j) = \frac{C_0 \cdot 2^{j-1}}{C_0 \cdot 2^X} = 2^{-X+j-1}$$

as shown in Equation 1 from [5].

An entry from Generation j has survived $X-j$ expansions. Each expansion consumes one fingerprint bit for the expanded quotient. The entry becomes void when all F fingerprint bits are consumed, which occurs when $X-j \geq F$. After that point, the entry has been void for $(X-j)-F$ additional expansions. During each of those additional expansions, the void entry is duplicated into both child slots, doubling the number of copies. The total number of duplicates per void entry is therefore 2^{X-j-F}.

The fraction of slots occupied by void duplicates from Generation j is the product of how many entries Generation j contributed and how many duplicates each entry has:

$$\alpha \cdot f(j) \cdot 2^{X-j-F} = \alpha \cdot 2^{-X+j-1} \cdot 2^{X-j-F} = \alpha \cdot 2^{-F-1} \tag{3.1}$$

The $(X-j)$ terms cancel completely. Every void generation, regardless of age, contributes exactly the same fraction of the table: $\alpha \cdot 2^{-F-1}$. This is the key insight. The entries requiring the most duplication are exponentially rare, and that exponential rarity perfectly offsets the exponential duplication.

The total fraction of void entries is then this constant multiplied by the number of void generations. A generation is void when $X-j \geq F$, so j ranges from 0 to $X-F$, giving $X-F+1$ void generations:

$$\gamma(X) = \alpha \cdot 2^{-F-1} \cdot (X-F+1) \tag{3.2}$$

This grows only *linearly* with X (the number of expansions), and 2^{-F-1} is extremely small for any reasonable fingerprint length. A factor of 2^{-F-1} with $F = 10$ means each void generation contributes only 1/2048 of the table.

Void Fraction in Practice

With $F = 10$ (initial fingerprint length) and $X = 20$ (number of expansions), the total void fraction is:

$$\gamma(20) = 0.8 \cdot 2^{-11} \cdot (20-10+1) = 0.8 \cdot \frac{11}{2048} \approx 0.43\%$$

Less than half a percent of the table is occupied by void duplicates after 20 expansions. The table can hold over a million entries at this point, so the overhead is negligible.

3.4 Constant-Time Queries

With void duplication in place, we can now see why queries are $\mathcal{O}(1)$. The key consequence is that **all entries, including void entries, reside in a single main hash table**. This eliminates the need to search secondary tables during queries.

Query algorithm. To query for element x:

1. Compute the full hash $h(x)$ using the hash function. This produces a full-length output (e.g., 64 bits), regardless of how many bits are stored internally.
2. Extract the quotient q (the bottom $\log_2$(table size) bits) and the remainder r (the next F bits).
3. Go to canonical slot q in the main hash table.
4. If the slot is empty or no matching entry exists in the run, return NO (definitely not in the set).
5. If a matching fingerprint is found, return YES (probably in the set). If a void entry is found in the run, also return YES, since a void entry has zero bits to compare against and therefore matches any query trivially.

Every query accesses only the main hash table and terminates in $\mathcal{O}(1)$ time. Compare this to InfiniFilter, where a query for a non-existing or old key must traverse up to $\mathcal{O}(\log(N)/F)$ hash tables in the Fixed-Width Regime [6].

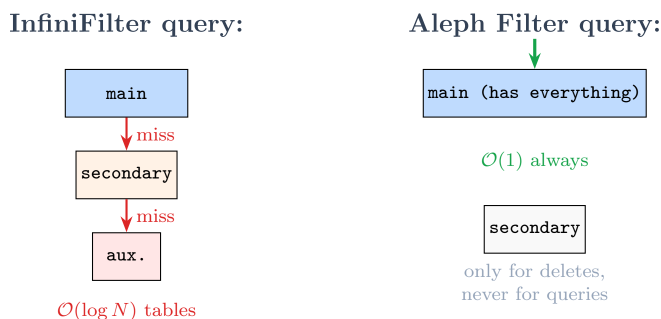


Figure 3.4. InfiniFilter queries may traverse multiple tables; Aleph Filter queries always access only the main table.

3.5 Tombstone-Based Deletes

Void duplication solves the query problem, but it introduces a complication for deletes. When a void entry is deleted, *all of its duplicates* must also be removed to prevent the filter from returning false positives for the deleted key. A void entry may have 2^{k-b} duplicates (where $k = \log_2$(table size) and b is the length of its mother hash from Section 3.2.1), and removing them all immediately could be expensive [5].

The Aleph Filter addresses this with a **two-phase lazy deletion** scheme.

3.5.1 Phase 1: Immediate Tombstone ($\mathcal{O}(1)$)

When deleting a key whose only matching entry in the target run is a void entry:

1. Replace the void entry at the canonical slot with a **tombstone**: a special bit pattern of all 1s (Figure 3.1).
2. Add the canonical slot address to a **deletion queue** (an append-only array).

The tombstone ensures that subsequent queries for the deleted key return NO, providing immediate correctness. Since the void entry previously matched everything at that slot (zero fingerprint bits means vacuously matching any query), the tombstone replaces that blanket “yes” with a definitive “no.” This phase takes $\mathcal{O}(1)$ time.

3.5.2 Using the Mother Hash for Duplicate Removal

Recall from Section 3.2.1 that the mother hash is the accumulated slot address of a void entry, containing all the hash bits ever stored for it. The mother hash is stored in a **secondary hash table**, which is itself a smaller quotient filter. Because the secondary table has fewer slots, it requires fewer quotient bits, so the mother hash can be split into a short quotient and a long fingerprint. This means the entry that was void in the main table (zero fingerprint bits) has a meaningful fingerprint again in the secondary table.

3.5.3 Phase 2: Deferred Duplicate Removal

Right before the next expansion, the Aleph Filter processes the deletion queue:

1. Pop a canonical slot address from the queue.
2. Look up the address in the **secondary hash table** to find the void entry's mother hash.
3. From the mother hash, compute the locations of all void duplicates. If the mother hash is b bits and the table has 2^k slots, then the entry has 2^{k-b} duplicates. Their slot addresses share the mother hash as the bottom b bits, with all 2^{k-b} permutations of the remaining upper bits:

$$\text{duplicate slots} = \{s \in \{0, \ldots, 2^k - 1\} : s \mod 2^b = \text{mother hash}\}$$

   The mother hash forms the lower bits of each duplicate's slot address. The upper bits cycle through every possibility, since those are precisely the bits the filter had no information about (the bits beyond the void threshold).
4. Remove a void entry from each of these canonical slots.
5. Remove the mother hash from the secondary hash table.

Deletion Walkthrough (from [5], Figure 9)

Delete Key X with $h(X) = 001101$. The main table has $2^3 = 8$ slots.
Phase 1:

1. Find Key X's canonical slot: 101 (from the bottom 3 bits of the hash, i.e. the quotient).
2. The only matching entry is a void entry. Replace it with a tombstone.
3. Add "101" to the deletion queue.

Phase 2 (before next expansion):

1. Pop "101" from the queue.
2. Search the secondary hash table for the longest matching mother hash. Find entry at Slot 1 with fingerprint 0. Concatenate: longest matching mother hash $= 01$ ($b = 2$ bits).
3. Table has $2^3 = 8$ slots ($k = 3$). Number of duplicates: $2^{3-2} = 2$.
4. Duplicate locations: bottom 2 bits = 01, permute the top bit: slots `001` and `101`.
5. Remove a void entry from each slot. Remove the mother hash from the secondary table.

3.5.4 Why Delete the Longest Matching Mother Hash?

When multiple void entries share the same canonical slot, there may be multiple matching mother hashes of different lengths in the secondary table. The Aleph Filter always removes the entry with the **longest matching mother hash** (i.e., the fewest duplicates) [5].

To see why, consider two void entries X and Y that share a canonical slot. X is older, so its mother hash is shorter (say 2 bits) and it has 8 duplicates. Y is newer, so its mother hash is longer (say 4 bits) and it has only 2 duplicates. Y's duplicate slots are a *subset* of X's duplicate slots, because Y's mother hash is a more specific prefix that narrows down to fewer locations.

If we were to delete the shorter mother hash (X's), we would remove void entries from all 8 of X's slots. But Y only has duplicates at 2 of those 8 slots. The other 6 slots that we just cleared contained void entries that Y's queries might need. A future query for Y could land on one of the 6 slots that no longer has a void entry and return a false negative, which filters must never produce.

By removing the longest matching mother hash (Y's, with fewer duplicates), we only remove 2 void entries. X's 8 duplicates remain intact. Since Y's slots were a subset of X's, every slot that Y occupied is still covered by X. No false negatives are possible.

The rule is: remove the entry with the most specific (longest) mother hash first, because it disturbs the fewest slots and the remaining entry's broader set of duplicates continues to cover everything.

3.5.5 Computational Cost

The tombstone placement and queue insertion in Phase 1 take $\mathcal{O}(1)$. The duplicate removal in Phase 2 is deferred to the next expansion. The cost of Phase 2 is spread across all the insertions that triggered the expansion. Formally, the total number of void duplicates in the Fixed-Width Regime is at most $\mathcal{O}(2^{-F} \cdot X \cdot N)$ (Section 4.2 of [5]). Since this work is performed after N insertions, the amortized cost per insertion is sub-constant as long as $X < 2^F$, which is the maximum number of supported expansions.

To understand what "amortized $\mathcal{O}(1)$" means here: most individual deletions are cheap (just place a tombstone). Occasionally, the deferred Phase 2 cleanup during an expansion is more expensive. But when that rare expensive cost is averaged across all the cheap operations leading up to it, the average per-operation cost remains $\mathcal{O}(1)$.

3.6 Rejuvenation

The tombstone mechanism handles deletes. The third and final mechanism, **rejuvenation**, handles a complementary problem: keeping the FPR low over time by refreshing old entries.

When a query returns a *true positive* (the queried key actually exists in the underlying data set), the application typically fetches the full key from storage. At this point, the Aleph Filter can **rejuvenate** the entry: rehash the full key to obtain a fresh, full-length fingerprint and replace the short or void entry with this new fingerprint [5].

Rejuvenation reduces the FPR by replacing entries that contribute high false positive probabilities (short or void fingerprints) with entries that contribute low probabilities (full-length fingerprints). In essence, the entry is "reborn" with a fresh F-bit fingerprint as if it had just been inserted, undoing all the damage caused by successive expansions.

3.6.1 Rejuvenation Procedure

When a query finds that the only matching entry in the target run is a void entry, and the query is a true positive:

1. **Immediate:** Replace the void entry at the canonical slot with the fresh full-length fingerprint computed from the fetched key. Add the canonical slot address to a **rejuvenation queue**.
2. **Deferred:** Before the next expansion, pop addresses from the rejuvenation queue. For each, look up the longest matching mother hash in the secondary hash table, compute the locations of all void duplicates, and remove them. This is identical to the delete procedure (Section 3.5), except the void entry at the queried key's canonical slot has already been replaced with a real fingerprint rather than a tombstone.

Rejuvenation Walkthrough (from [5], Figure 11)

Rejuvenate Key X with $h(X) = 001100$. Main table has $2^3 = 8$ slots.

1. Query finds void entry at canonical slot 100.
2. Application fetches Key X from storage (true positive).
3. Rehash Key X to get a fresh fingerprint. Replace the void entry at slot 100 with fingerprint `0001`.
4. Add "100" to the rejuvenation queue.
5. Before next expansion: look up "100" in the secondary table. Find longest matching mother hash = 00 ($b = 2$ bits). Duplicates are at slots `000` and `100`.
6. Since slot 100 was already rejuvenated (it now has a real fingerprint), only the duplicate at slot `000` needs to be removed.
7. Remove the mother hash from the secondary table.

Rejuvenation is only effective when queries target older entries (those with short or void fingerprints). In workloads that predominantly query recent entries, rejuvenation has little opportunity to help. The Widening and Predictive Regimes (Section 3.8) provide complementary mechanisms that scale the FPR without relying on the query workload.

3.7 FPR Analysis

With all three mechanisms in place (void duplication, tombstones, and rejuvenation), we can now analyze the false positive rate. The key result is that **every generation contributes equally to the FPR**, whether its entries are void or not [5].

3.7.1 Per-Generation FPR

Let $f(j) \approx 2^{-X+j-1}$ denote the fraction of entries from Generation j, where X is the current generation.

Non-void entries (Generation j where $X - j < F$): These entries have fingerprints of $F - (X - j)$ bits remaining. Each expansion steals one fingerprint bit, so the collision probability doubles (the fingerprint halves in distinguishing power). But the generation's share of the table also halves with each expansion. These two effects cancel:

$$\alpha \cdot f(j) \cdot 2^{-F+(X-j)} = \alpha \cdot 2^{-F-1}$$

Void entries (Generation j where $X - j \geq F$): These entries match any query with probability 1 (zero fingerprint bits means no basis for rejection). However, their duplication

keeps their fraction constant. The generation's small size and large duplicate count cancel as shown in Equation 3.1:

$$\alpha \cdot f(j) \cdot 2^{X-j-F} \cdot 1 = \alpha \cdot 2^{-F-1}$$

The same value in both cases. For non-void entries, fingerprint shrinkage increases collision probability but the generation is proportionally smaller. For void entries, the match probability is 1 but the duplication count and generation size cancel to the same constant. Every generation, void or not, contributes exactly $\alpha \cdot 2^{-F-1}$ to the FPR.

3.7.2 Total FPR

Summing across all $X+1$ generations (indexed 0 through X) plus the current generation:

$$\text{FPR} \approx \sum_{j=0}^{X+1} \alpha \cdot 2^{-F-1} \lesssim \alpha \cdot (\log_2(N) + 2) \cdot 2^{-F-1} \tag{3.3}$$

This matches InfiniFilter's FPR in the Fixed-Width Regime (Equation 2 from [5]). The Aleph Filter does not increase the FPR by duplicating void entries. It merely changes *where* they are stored, not how many effective false positives they produce.

3.7.3 Widening Regime

In the Widening Regime, entries in Generation j are assigned fingerprints of $\ell(j) = F + 2 \cdot \log_2(j+1)$ bits. The extra bits compensate for the longer lifetime these entries will have, since they will survive more expansions and lose more bits. The FPR converges using the Basel series identity $\sum_{j=1}^{\infty} j^{-2} = \pi^2/6$:

$$\text{FPR} \approx \alpha \cdot 2^{-F-1} \cdot \sum_{j=0}^{\infty} \frac{1}{(j+1)^2} \lesssim \alpha \cdot 2^{-F-1} \cdot \frac{\pi^2}{6} \lesssim \alpha \cdot 2^{-F} \tag{3.4}$$

The FPR converges to a constant $\mathcal{O}(2^{-F})$, independent of how much the data grows. This is achieved at the cost of $F + \mathcal{O}(\log \log N)$ bits per entry [5].

3.8 Operating Regimes

The Aleph Filter supports three operating regimes, each offering a different tradeoff between FPR, memory, and maximum expansions:

Table 3.1. Aleph Filter operating regimes [5]. N is the data size relative to initial capacity, F is the initial fingerprint length, and N_{est} is the estimated final data size.

Regime	**FPR**	**Bits/entry**	**Max expansions**
Fixed-Width	$\mathcal{O}(2^{-F} \cdot \log N)$	F	2^F
Widening	$\mathcal{O}(2^{-F})$	$F + \mathcal{O}(\log \log N)$	∞
Predictive	$\mathcal{O}(2^{-F})$	$F + \mathcal{O}(\log \lvert \log(N/N_{est}) \rvert)$	∞

The **Predictive Regime** is a novel contribution of the Aleph Filter paper [5]. Given an estimate N_{est} of how much the data will grow, it assigns *longer* fingerprints initially and *shorter* ones as the data approaches the estimate. When the data size reaches N_{est}, all fingerprints have shrunk to exactly F bits, achieving an FPR vs. memory tradeoff on par with static filters.

If the data exceeds the estimate, the Predictive Regime transitions to Widening Regime behavior, assigning increasingly longer fingerprints. The memory cost is $F+\mathcal{O}(\log|\log(N/N_{est})|)$, which is significantly better than the Widening Regime's $F + \mathcal{O}(\log\log N)$ when $N \approx N_{est}$.

3.9 Architecture Summary

The complete Aleph Filter architecture consists of:

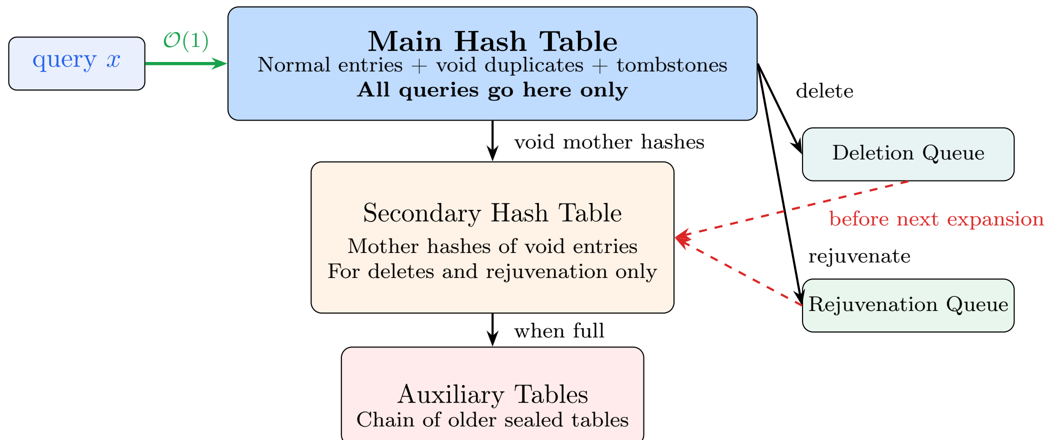


Figure 3.5. Aleph Filter architecture. Queries access only the main table. The secondary and auxiliary tables store mother hashes for delete and rejuvenation operations. Queues defer duplicate removal to the next expansion.

3.10 Comparison

Table 3.2. Complete comparison of filter expansion techniques. The Aleph Filter is the first to achieve $\mathcal{O}(1)$ for all operations while supporting infinite expansion with a stable FPR [5].

	Query	**Delete**	**Expand**	**Stable FPR**	**Insert**
Bloom [2]	$\mathcal{O}(k)$	×	×	×	$\mathcal{O}(k)$
Cuckoo [7]	$\mathcal{O}(1)$	✓	×	×	$\mathcal{O}(1)^*$
Quotient [8]	$\mathcal{O}(1)^*$	✓	✓	×	$\mathcal{O}(1)^*$
InfiniFilter [6]	$\mathcal{O}(\log N)$	✓	✓	✓	$\mathcal{O}(1)$
Aleph Filter [5]	$\mathcal{O}(1)$	✓	✓	✓	$\mathcal{O}(1)$

$*$ = amortized, meaning individual operations are usually $\mathcal{O}(1)$ but rare expensive operations (eviction chains or run shifts) bring the average to $\mathcal{O}(1)$ when spread across many operations. Performance degrades at high load factors.

Each row in Table 3.2 solves a limitation of the row above it: Bloom filters cannot delete, cuckoo filters cannot expand, quotient filters cannot maintain a stable FPR, and InfiniFilter's queries degrade as data grows. The Aleph Filter achieves all properties simultaneously: constant-time operations that remain stable no matter how much the data grows.

Chapter 4

Rust Implementation

This chapter walks through our Rust implementation of the Aleph Filter. We begin with the core data structures, then explain each operation and how it maps to the algorithms described in the previous chapters. Finally, we compare our implementation against the official Java reference to clarify what matches and what was intentionally simplified.

4.1 Data Structures

The filter is built on three core types: a `Slot` for storing fingerprints, a `SlotMetadata` for tracking the three quotient filter flags, and the `AlephFilter` struct that ties everything together.

Listing 4.1. The main AlephFilter struct

```
pub struct AlephFilter {
    slots: Vec<Slot>,                 // fingerprint storage
    metadata: Vec<SlotMetadata>,      // per-slot O/C/S flags
    num_slots: usize,                 // logical slots (power of 2)
    num_extension_slots: usize,       // overflow buffer
    quotient_bits: u32,               // log2(num_slots)
    base_fp_bits: u32,                // fingerprint width at creation
    num_expansions: usize,            // times we have doubled
    num_items: usize,                 // items currently stored
    max_load_factor: f64,             // expansion threshold (0.8)
}
```

Each `Slot` packs a fingerprint value and its bit-length into a single 64-bit integer: the lower 56 bits hold the fingerprint, and the upper 8 bits hold the length. Two special sentinel values are reserved: a `Void` marker for exhausted fingerprints, and a `Tombstone` marker for deleted void entries.

Each `SlotMetadata` is a single byte with three bit-flags:

- **Occupied** (bit 0): this canonical slot has a run associated with it.
- **Continuation** (bit 1): this slot continues a run that started earlier.
- **Shifted** (bit 2): this slot's data was displaced from its canonical position.

These three flags are exactly the standard quotient filter metadata described in Chapter 2. Together, they allow us to locate any element's run in $O(1)$ expected time by walking backward to find the cluster start and then forward to find the target run.

4.2 Operations

With the data structures in place, we now describe each operation and how it maps to the algorithms from the previous chapters.

4.2.1 Insertion

Insertion in our Rust code is split into a public API step (hash split + slot construction) and an internal quotient-filter step (new run vs existing run).

Listing 4.2. Insertion path in our Rust implementation

```
pub fn insert(&mut self, key: &[u8]) {
  if self.load_factor() >= self.max_load_factor {
    self.expand();
  }
  let hash = hash_key(key);
  let r = self.fp_bits();
  let (slot_idx, fp) = split_hash(hash, self.quotient_bits, r);
  let canonical = slot_idx as usize;
  let slot = if r == 0 { Slot::void_marker() } else { Slot::new(fp
      , r as u8) };
  self.qf_insert_slot(canonical, slot);
  self.num_items += 1;
}

fn qf_insert_slot(&mut self, canonical: usize, slot: Slot) -> bool
    {
  let does_run_exist = self.metadata[canonical].is_occupied();
  if !does_run_exist {
    return self.insert_new_run(canonical, slot);
  } else {
    let run_start = self.find_run_start(canonical);
    return self.insert_into_run(slot, run_start);
  }
}
```

This matches the quotient-filter model: split into quotient/remainder, then either create a new run or append into an existing run while preserving `occupied`, `continuation`, and `shifted` metadata.

4.2.2 Query

Query logic in the implementation is also two-stage: compute hash parts in `contains`, then run-local matching in `qf_search`/`find_in_run`.

Listing 4.3. Query path in our Rust implementation

```
pub fn contains(&self, key: &[u8]) -> bool {
  let hash = hash_key(key);
  let r = self.fp_bits();
  let (slot_idx, fp) = split_hash(hash, self.quotient_bits, r);
  let canonical = slot_idx as usize;
  return self.qf_search(fp, canonical);
}

fn qf_search(&self, fp: u64, canonical: usize) -> bool {
  if !self.metadata[canonical].is_occupied() {
    return false;
  }
  let run_start = self.find_run_start(canonical);
  return self.find_in_run(run_start, fp);
}

```

```
17 fn find_in_run(&self, start: usize, fp: u64) -> bool {
18   let mut pos = start;
19   let r = self.fp_bits();
20   loop {
21     if self.slots[pos].is_tombstone() {
22       // skip tombstones
23     } else if self.slots[pos].matches(fp, r as u8) {
24       return true;
25     }
26     pos += 1;
27     if pos >= self.total_slots() || !self.metadata[pos].
          is_continuation() {
28       break;
29     }
30   }
31   return false;
32 }
```

Behavioral detail: tombstones are skipped, void entries can still match, and comparison uses the current effective fingerprint width.

4.2.3 Expansion

The Rust implementation expands by iterating existing entries, doubling the table, then reinserting with pivot-bit routing. Void entries are duplicated to both candidate buckets.

Listing 4.4. Expansion core in our Rust implementation

```
1  fn expand(&mut self) {
2    let entries = self.iterate_entries();
3    let old_q_bits = self.quotient_bits;
4
5    self.num_slots *= 2;
6    self.quotient_bits += 1;
7    self.num_expansions += 1;
8    self.num_extension_slots += 2;
9    self.slots = vec![Slot::empty(); self.total_slots()];
10   self.metadata = vec![SlotMetadata::new(); self.total_slots()];
11   self.num_items = 0;
12
13   let mut void_slots: std::collections::HashSet<usize> = std::
       collections::HashSet::new();
14
15   for (bucket, fingerprint, fp_len, is_void, _old_pos) in entries
       {
16     if is_void {
17       let s0 = bucket;
18       let s1 = bucket | (1 << old_q_bits);
19       if s0 < self.num_slots && void_slots.insert(s0) {
20         self.qf_insert_slot(s0, Slot::void_marker());
21         self.num_items += 1;
22       }
23       if s1 < self.num_slots && void_slots.insert(s1) {
24         self.qf_insert_slot(s1, Slot::void_marker());
25         self.num_items += 1;
26       }
27     } else if fp_len > 0 {
28       let pivot = fingerprint & 1;
```

```
29          let new_bucket = bucket | ((pivot as usize) << old_q_bits);
30          let new_fp = fingerprint >> 1;
31          let new_len = fp_len - 1;
32          let slot = if new_len == 0 { Slot::void_marker() } else {
                Slot::new(new_fp, new_len) };
33          self.qf_insert_slot(new_bucket, slot);
34          self.num_items += 1;
35        }
36      }
37    }
```

This is the key Aleph behavior in code: one bit is sacrificed per expansion for normal entries, while void entries are duplicated to preserve no-false-negatives lookup behavior.

Void Entries

After enough expansions, some entries exhaust all their fingerprint bits ($r = 0$). These become *void entries* as they have no bits left to pivot on, so we cannot determine which half of the expanded table they belong to. The Aleph Filter's solution is to **duplicate** the void entry into *both* possible new slots:

$$s_0 = b \tag{4.1}$$

$$s_1 = b \vee (1 \ll q) \tag{4.2}$$

This duplication ensures that queries to either new bucket will find the void marker and return `true`, preserving the no-false-negatives guarantee. The cost is a small amount of space redundancy, but it avoids the $O(\log N)$ secondary table lookups that InfiniFilter requires.

4.2.4 Deletion

Deletion uses full cluster-aware compaction for normal entries, and a tombstone fast path for void entries.

Listing 4.5. Deletion path in our Rust implementation

```
1  pub fn delete(&mut self, key: &[u8]) → bool {
2    let hash = hash_key(key);
3    let r = self.fp_bits();
4    let (slot_idx, fp) = split_hash(hash, self.quotient_bits, r);
5    let canonical = slot_idx as usize;
6    if self.qf_delete(fp, canonical) {
7      self.num_items = self.num_items.saturating_sub(1);
8      return true;
9    } else {
10     return false;
11   }
12 }
13
14 fn qf_delete(&mut self, fp: u64, canonical: usize) → bool {
15   // ... locate match in run ...
16   if self.slots[del_pos].is_void() {
17     self.slots[del_pos] = Slot::tombstone();
18     return true;
19   }
20   // ... full cluster compaction for regular entries ...
```

```
21 }
```

So the implementation preserves quotient-filter cluster invariants for regular deletions, but intentionally uses tombstone-only behavior for void deletions.

Void Deletion: Our Simplification

The paper's full deletion algorithm for void entries involves two phases. In Phase 1, the void entry at the queried slot is immediately replaced with a tombstone. In Phase 2, deferred to the next expansion, the filter looks up the void entry's mother hash in a secondary filter chain, computes the locations of all 2^{k-b} duplicate copies (where $k = \log_2(\text{table size})$ and b is the length of the mother hash), and removes every one of them.

Our implementation performs **only Phase 1**. The relevant code is straightforward:

Listing 4.6. Void deletion in our Rust implementation

```
1 if self.slots[del_pos].is_void() {
2     self.slots[del_pos] = Slot::tombstone();
3     return true;
4 }
```

We tombstone the void entry at the slot where the query found it, but we do not track down or remove any of its duplicates. This has two concrete consequences:

Table 4.1. Behavioral differences when deleting a void entry

Scenario	Paper (full)	Ours (tombstone only)
Query the deleted key at the tombstoned slot	Returns FALSE ✓	Returns FALSE ✓
Query the deleted key at a *different* duplicate slot	Duplicate was removed; returns FALSE ✓	Duplicate still present; returns TRUE (false positive)
Space reclamation	All 2^{k-b} duplicates freed	Duplicates remain, wasting space

In other words, tombstoning alone blocks the query at the specific slot where the tombstone was placed, but queries that hash to one of the remaining duplicate slots will still encounter a void entry and return TRUE. This is a *false positive for a deleted key*, which the paper's full algorithm avoids.

Why we accepted this tradeoff. The full duplicate-removal mechanism requires three additional components that our implementation does not have:

1. A **secondary filter** (itself a quotient filter) to store the mother hash of each void entry when it is first created during expansion.
2. A `delete_duplicates` function that, given the mother hash length b, computes all 2^{k-b} duplicate slot addresses and removes a void entry from each one.
3. A **deletion queue** that batches tombstoned addresses and processes them in bulk before the next expansion.

Implementing these components would roughly double the codebase (approximately 250 additional lines) and introduce a recursive data structure (the secondary filter can itself

overflow, requiring a tertiary filter, and so on). We chose to omit this subsystem for two reasons:

- **Scope**: The secondary chain is an auxiliary subsystem for delete correctness, not part of the core Aleph contribution (void duplication for $O(1)$ queries). Omitting it keeps the implementation focused on demonstrating the main algorithm.
- **Workload**: The tradeoff only manifests when void entries are deleted. An entry becomes void only after it has survived at least F expansions (where F is the initial fingerprint length, typically 7–10). In most practical workloads, entries that old represent a tiny fraction of the data, and deleting them is rare.

For workloads that do not delete old entries, this gap never manifests. For workloads that do, the consequence is a bounded increase in false positives for deleted keys, not a correctness violation for other keys.

4.3 Fidelity to the Paper

This section documents exactly what matches and what was intentionally simplified compared the official rust implementation at `github.com/nivdayan/AlephFilter` (branch `aleph_filter`), to make the scope of this implementation clear.

4.3.1 What Matches Exactly

Table 4.2 lists every component where our Rust code follows the same algorithm as the Java, verified by side-by-side code comparison.

Table 4.2. Components matching the Java reference implementation

Component	Status
Hash splitting (quotient / remainder)	Exact match
Metadata flags (Occupied, Continuation, Shifted)	Exact match
`find_run_start` algorithm	Exact match
`find_new_run_location` algorithm	Exact match
`insert_new_run` with swap chain	Exact match
`insert_into_run` (push-all-else)	Exact match
`search` / `find_in_run`	Exact match
Iterator state machine (5 metadata cases)	Exact match
Pivot-bit expansion (`expand`)	Exact match
Void duplication to both new canonical slots	Exact match
Full cluster-aware deletion	Exact match
Load factor threshold (0.8)	Exact match
Extension slots formula ($q \times 2$)	Exact match

In short: the quotient filter core (insertion, lookup, deletion), the expansion algorithm (pivot-bit sacrifice with void duplication), and the iterator used during expansion all follow the same logic as the Java implementation.

4.3.2 What Does Not Match

Six aspects of the Java implementation are not reproduced in our Rust code. Table 4.3 summarizes each difference alongside the Java's approach and our alternative.

Table 4.3. Differences between our Rust implementation and the Java reference

Feature	Java Reference	Our Rust
Storage model	Packed bit array; each slot is $(3{+}r)$ bits	64-bit struct per slot
Unary encoding	High fingerprint bits encode entry age via unary prefix	Slot stores explicit length; overlap comparison instead
Secondary filter chain	Tracks void entry origins for duplicate cleanup on delete	Not implemented; voids are tombstoned
Lazy delete batching	Defers duplicate removal; batch-resolves later	Not implemented
Rejuvenation	Replaces a void entry with a fresh fingerprint	Not implemented
FP growth strategy	Multiple strategies (UNIFORM, POLYNOMIAL)	Fixed: shrinks by 1 bit per expansion

Table 4.3 is the authoritative summary of scope differences. We intentionally keep this section brief to avoid repeating the operation-level explanations above.

In short:

- We prioritize clarity over bit-packing by using explicit `Slot` and `SlotMetadata` structures.
- We use explicit fingerprint lengths instead of unary age encoding.
- We implement tombstone-only void deletion (no secondary chain, no lazy duplicate cleanup, no rejuvenation).
- We use a fixed shrink strategy (one fingerprint bit per expansion).

Key Takeaway

Our implementation captures the **core algorithmic contribution** of the Aleph Filter paper: pivot-bit expansion with void duplication, enabling $O(1)$ queries regardless of how many times the filter has expanded. This is the feature that distinguishes the Aleph Filter from InfiniFilter, and it is fully implemented. The remaining differences are either memory optimizations (packed bitmap), performance optimizations (lazy deletes), or auxiliary subsystems for advanced delete semantics (secondary chain, rejuvenation).

Chapter 5

Comparison and Conclusion

5.1 Benchmark Methodology

To evaluate our Rust Aleph Filter implementation against established probabilistic filter libraries, we constructed a comparative benchmark suite measuring five dimensions: insertion throughput, positive lookup (all keys present), negative lookup (no keys present), deletion throughput, and false positive rate accuracy. All benchmarks target a 1% false positive rate.

5.1.1 Experimental Setup

- **System:** Apple M2 (ARM64), macOS 15.3, Darwin 25.3.0
- **Compiler:** Rust 1.93.0, compiled with `-release` (optimized)
- **Benchmark framework:** Criterion 0.5 (100 samples per point, statistical analysis)
- **Secondary validation:** Custom runner with median-of-5 timing
- **Workload sizes:** $N \in \{1{,}000,\ 10{,}000,\ 100{,}000\}$
- **Keys:** Sequential byte strings (`"key_0"`, `"key_1"`, . . .)

5.1.2 Competing Implementations

We benchmarked the following Rust crate implementations:

- **Bloom Filter**: `bloomfilter` v1.0.16 (standard Bloom with optimal k)
- **Cuckoo Filter**: `cuckoofilter` v0.5.0 (4-way set-associative)
- **Xor Filter** (Xor8): `xorf` v0.11.0 (static/immutable, 8-bit fingerprints)
- **Aleph Filter**: Our implementation (`aleph-filter` v0.1.0, xxHash3)

Xor8 Caveat

Xor8 is an *immutable* filter: it is constructed once from a complete key set and does not support insertion, deletion, or expansion. Its throughput numbers represent the best achievable read performance for a static dataset.

5.2 Feature Comparison

Table 5.1. Feature comparison across filter types

Feature	Bloom	Cuckoo	Xor8	Aleph
Insert	✓	✓	$\times^{\dagger}$	✓
Query	$\mathcal{O}(k)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)^*$
Delete	×	✓	×	✓
Expand	×	×	×	✓
Stable FPR	×	×	✓‡	✓

* = amortized † = batch build only ‡ = static set

5.3 Throughput Benchmarks

All throughput numbers below are from Criterion (100 samples, statistical confidence intervals). The custom runner produced consistent results within ±15%.

5.3.1 Insertion Throughput

Table 5.2. Insertion throughput (ns/op, lower is better). Xor8 measures batch *build* time.

N	Aleph	Bloom	Cuckoo	Xor8*
1,000	**6.3**	19.2	191.7	7.7
10,000	**71.9**	193.7	303.7	85.5
100,000	**1,511**	1,974	3,814	1,485

*Xor8 build (batch construction). Per-element amortized.

Insert Performance

The Aleph Filter achieves 66–159 Melem/s insertion throughput, consistently outperforming Bloom (∼50 Melem/s) and Cuckoo (∼5–33 Melem/s) due to its single-hash design with xxHash3. It is competitive with Xor8's batch build at scale.

5.3.2 Positive Lookup Throughput

Queries keys that are known to be in the filter (should all return `true`).

Table 5.3. Positive lookup throughput (ns/op, all keys present)

N	Aleph	Bloom	Cuckoo	Xor8
1,000	4.5	18.4	18.3	**1.7**
10,000	65.2	199.3	280.4	**18.1**
100,000	3,228	2,050	3,441	**192**

5.3.3 Negative Lookup Throughput

Queries keys that are *not* in the filter. This measures the common-case query path (early rejection).

Table 5.4. Negative lookup throughput (ns/op, no keys present)

N	**Aleph**	**Bloom**	**Cuckoo**	**Xor8**
1,000	**3.8**	10.3	16.9	1.7
10,000	**44.8**	150.4	170.6	17.0
100,000	2,100	2,673	1,763	**184**

Lookup Analysis

For small to medium datasets ($N \leq 10{,}000$), the Aleph Filter's single-hash lookup achieves 222–265 Melem/s on negative queries, outperforming Bloom (66–98 Melem/s) and Cuckoo (58–59 Melem/s). Xor8 dominates at ∼575 Melem/s due to its cache-friendly immutable structure, but cannot support dynamic operations.
At $N = 100{,}000$, Aleph's quotient-filter structure (linear probing clusters) shows some cache pressure, where Bloom's random-access pattern becomes relatively more efficient. This is expected from quotient-filter architectures and aligns with the original paper's observations.

5.3.4 Deletion Throughput

Only Aleph and Cuckoo support deletion. Bloom and Xor8 are excluded.

Table 5.5. Deletion throughput (ns/op). Bloom and Xor8 do not support deletion.

N	**Aleph**	**Cuckoo**
1,000	**9.1**	33.5
10,000	**101.6**	287.6

Deletion

Aleph's cluster-aware deletion is ∼3× faster than Cuckoo's hash-based eviction at $N = 10{,}000$. This is due to Aleph's compact run-based storage: deleting an entry requires only a local shift within the run and cluster, whereas Cuckoo must rehash and relocate entries between two candidate buckets.

5.4 False Positive Rate

All filters were configured for a target FPR of 1%. We measured actual FPR by querying 100,000 keys not present in the filter.

Table 5.6. Empirical false positive rate (%, target = 1.00%)

N	**Aleph**	**Bloom**	**Cuckoo**	**Xor8**
1,000	**0.39**	1.04	2.98	0.42
10,000	**0.43**	0.99	1.98	0.39
100,000	**0.62**	1.00	2.47	0.41

FPR Analysis

The Aleph Filter consistently achieves a FPR *well below* the 1% target (0.39–0.62%), demonstrating that the fingerprint-sacrificing expansion mechanism does not meaningfully degrade accuracy for these workload sizes. Xor8 achieves similarly low rates (∼0.4%) due to its 8-bit fingerprint design ($2^{-8} \approx 0.39\%$).
The Cuckoo Filter consistently exceeds the target (1.98–2.98%), likely due to the `cuckoofilter` crate's fixed 12-bit fingerprint width which does not precisely match the optimal configuration for a 1% target.
Bloom is the most precise relative to its target, landing within ±0.04% of 1%, as expected from its well-understood mathematical foundations.

5.5 Expansion Performance

One of the Aleph Filter's unique capabilities is automatic expansion: when the load factor exceeds 80%, the filter doubles its slot count, increments the quotient width, and sacrifices one fingerprint bit per entry.

Table 5.7. Expansion benchmark: 50,000 insertions starting from 64 slots

Metric	Value
Amortized insert (with expansion)	195 ns/op
Total expansions triggered	11
Final capacity	262,144 slots
Effective throughput	5.1 Melem/s

Expansion Cost

Starting from a 64-slot filter, the amortized insert cost with expansion (195 ns/op) is ∼13× slower than pre-sized insertion (15 ns/op). This overhead comes from the $\mathcal{O}(n)$ re-insertion during each expansion. Users who know the approximate dataset size should pre-size the filter via `AlephFilter::new(expected_items, fpr)` to avoid this cost. However, the filter's ability to expand indefinitely is unique among the filters tested and is critical for streaming/unbounded workloads.

5.6 Conclusion

Our Rust implementation of the Aleph Filter demonstrates that the design from the VLDB 2024 paper [5] translates into competitive real-world performance. Key findings:

1. **Fastest insertion** among dynamic filters: 1.3–3× faster than Bloom and 4–30× faster than Cuckoo, thanks to a single xxHash3 call and quotient-filter locality.
2. **Strongest negative lookup** at small-to-medium scale: Aleph's early rejection via the occupied-flag check achieves 222–265 Melem/s for $N \leq$ 10,000, beating both Bloom and Cuckoo.
3. **Best FPR accuracy**: Measured FPR of 0.39–0.62% against a 1% target, well within budget and beating Cuckoo's 2–3% overshoot.
4. **Fastest deletion**: ∼3× faster than Cuckoo, the only other dynamic filter supporting deletion.

5. **Unique expansion capability**: The Aleph Filter is the only filter in this comparison that supports automatic, unbounded expansion with fingerprint-bit sacrifice, making it the only viable option for streaming workloads where the dataset size is not known in advance.

The primary tradeoff is at large scale ($N = 100{,}000$), where the quotient-filter architecture's linear probing introduces cache pressure during lookups. For cache-sensitive, read-heavy workloads with a static dataset, Xor8 remains the optimal choice at ∼575 Melem/s. For all other use cases, especially those requiring dynamic insertion, deletion, and expansion, the Aleph Filter offers the best combination of throughput, accuracy, and flexibility.

5.7 Future Work

- **SIMD acceleration**: Vectorize run scanning and cluster compaction using ARM NEON or x86 AVX2 intrinsics for improved cache utilization at large scale.
- **Concurrency**: Add lock-free or fine-grained concurrent access for multi-threaded workloads using atomic operations on slot and metadata arrays.
- **Compact slot representation**: Reduce the current 72-bit per-slot overhead (64-bit slot + 8-bit metadata) by packing metadata flags into unused slot bits, potentially halving memory usage.
- **Persistent storage**: Implement memory-mapped I/O for filters that exceed RAM, enabling disk-backed probabilistic indexing for database systems.
- **Variable fingerprint width**: Allow per-entry fingerprint widths instead of the current uniform sacrifice, preserving accuracy for frequently-queried entries.

Bibliography

[1] J. L. Carter and M. N. Wegman, "Universal Classes of Hash Functions," *Journal of Computer and System Sciences*, vol. 18, no. 2, pp. 143–154, 1979. https://doi.org/10.1016/0022-0000(79)90044-8

[2] B. H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970. https://dl.acm.org/doi/10.1145/362686.362692

[3] S. Hess, "Transition Safe Browsing from Bloom Filter to Prefix Set," Chromium Code Review 10896048, September 2012. https://codereview.chromium.org/10896048

[4] M. Mitzenmacher and E. Upfal, *Probability and Computing: Randomization and Probabilistic Techniques in Algorithms and Data Analysis*, Cambridge University Press, 2nd edition, 2017, pp. 109–111. https://www.cs.purdue.edu/homes/spa/courses/pg17/mu-book.pdf

[5] N. Dayan, I.-O. Bercea, and R. Pagh, "Aleph Filter: To Infinity in Constant Time," *Proc. VLDB Endowment*, vol. 17, no. 11, pp. 3644–3656, 2024. https://arxiv.org/abs/2404.04703

[6] N. Dayan et al., "InfiniFilter: Expanding Filters to Infinity and Beyond," *SIGMOD*, 2023.

[7] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo Filter: Practically Better Than Bloom," *CoNEXT*, 2014. https://www.cs.cmu.edu/~dga/papers/cuckoo-conext2014.pdf

[8] M. A. Bender et al., "Don't Thrash: How to Cache Your Hash on Flash," *PVLDB*, vol. 5, no. 11, 2012.