A Practical Adaptive Quotient Filter

by
David J. Lee

Shikha Singh, Advisor
Samuel McCauley, Co-Advisor

A thesis submitted in partial fulfillment of the requirements for the
Degree of Bachelor of Arts with Honors in Computer Science

Williams College
Williamstown, Massachusetts

May 29, 2021

Contents

1 Introduction **1**
1.1 Overview 2

2 Background **4**
2.1 Filters 4
2.1.1 Bloom filter 6
2.1.2 Cuckoo filter 7
2.1.3 Quotient filter 8
2.2 Adaptivity 10
2.2.1 Remote representation 11
2.2.2 Prior work on adaptive filters 11
2.3 Summary 13

3 Rank-and-Select Quotient Filter **14**
3.1 Rank and select 14
3.1.1 Definition 14
3.1.2 Fast implementation 15
3.2 Single-block RSQF 15
3.2.1 Membership queries 16
3.2.2 Insertions 18
3.2.3 Summary 19
3.3 Blocked RSQF 20
3.3.1 Offsets 20
3.3.2 Shifting metadata across blocks 22
3.3.3 Abstractions for rank and select 22
3.3.4 Queries 23
3.3.5 Insertions 24
3.4 Summary 24

4 Arithmetic Coding **26**
4.1 Introducing arithmetic coding 27
4.1.1 Encoding 27
4.1.2 Decoding 28
4.2 Fast approximate arithmetic codes 30
4.3 Summary 31

5 Practical Adaptive Quotient Filters **32**
5.1 Broom filter 32
5.1.1 Definition and remote representation 32
5.1.2 Fingerprint extensions 33

5.1.3 Core operations . . . 34
5.1.4 Bit reclamation . . . 35
5.2 Elements of a practical adaptive filter . . . 35
5.2.1 Adaptivity mechanism . . . 36
5.2.2 Arithmetic coding . . . 36
5.2.3 Space reclamation and rebuilds . . . 40
5.2.4 Remote representation . . . 40
5.3 Extension-based adaptive filter . . . 40
5.3.1 Architecture . . . 40
5.3.2 Insertions . . . 41
5.3.3 Queries . . . 41
5.3.4 Adapts . . . 43
5.3.5 Rebuilds . . . 45
5.4 Selector-based adaptive filter . . . 45
5.4.1 Architecture . . . 45
5.4.2 Insertions . . . 45
5.4.3 Queries . . . 46
5.4.4 Adapts . . . 46
5.4.5 Rebuilds . . . 48
5.5 Summary . . . 50

6 Experimental Results 51
6.1 Methodology . . . 51
6.1.1 Test parameters . . . 52
6.2 False positive rate tests . . . 53
6.2.1 FireHose streaming benchmarks . . . 53
6.2.2 Network traces . . . 55
6.2.3 Adversarial tests . . . 55
6.3 Throughput tests . . . 59
6.4 Summary . . . 62

7 Conclusion 63
7.1 Review of results . . . 63
7.2 Future directions . . . 63

List of Figures

2.1 Filters can cut out unnecessary but expensive queries early with high probability. Reproduced from [17]. 5
2.2 Without metadata, a table of remainders can be interpreted in multiple conflicting ways. 9
2.3 A simplified view of the Adaptive Cuckoo Filter. Elements in the remote state are updated in sync with fingerprints in the local state 12

3.1 Haswell instructions used to implement fast RANK and SELECT operations. 15
3.2 A singleton run and its interpretation. 16
3.3 Multiple runs and their interpretations. 16
3.4 RSQF notation. 16
3.5 Comparing the simple (unblocked) RSQF to a blocked RSQF. 21

4.1 An example of the encoding process. Reproduced in part from [21]. 28
4.2 Encoding a message m, visually. 29
4.3 Decoding the point 0.1 as the message "*aaa*". 29
4.4 Decoding an arithmetic code $p \in [0, 1)$ for a message of length l. We use the subroutine FINDSYMBOL to select the subinterval in the current interval that contains p. 30

5.1 Notation for fingerprint pieces. 34
5.2 Updating `low` in the optimized arithmetic code. 39
5.3 A block in the EXTQF's local filter L. 41
5.4 Defining DECODEEXT and DECODEBLOCK in terms of EXTQF metadata and DECODE. 44
5.5 A block in the SELQF. 46

6.1 FPR on FireHose active set data, 10 million lines. Each element is repeated 57 times on average. Lower is better. 53
6.2 FPR on FireHose power-law data, 50 million lines. Each element is repeated 584 times on average. Lower is better. 54
6.3 FPR results on CAIDA network trace data. 56
6.4 Adversary runs with initial Q/S values 10, 15, and 20. Each test was run for 10 iterations on each filter. The faded lines represent single iteratoins; the dark lines represent the average of 10 iterations. 57
6.5 Adversary runs with initial Q/S values 30 and 50. Each test was run for 10 iterations on each filter. The faded lines represent single iteratoins; the dark lines represent the average of 10 iterations. 58
6.6 Adversary results summary. FP Proportion captures the proportion of all elements in Q that have at least one false positive. 59
6.7 Insertion throughput over 2^{24} random insertions. Higher is better. 60
6.8 Query throughput on active set data, 10 million lines. Higher is better. 61

6.9 Query throughput results on power-law data, 1 billion lines. Higher is better. 61

List of Algorithms

3.1 Query algorithm for simple RSQF. [16] 17
3.2 Algorithm to find the first unused slot after i in an RSQF Q. [16] 18
3.3 Algorithm to insert an element into the simplified RSQF. [16] 19
3.4 A helper function that shifts offsets pointing to runends in $[a, b]$. 22
3.5 A helper function that shifts offsets pointing to i, the site of a remainder insertion for the quotient q. 22
3.6 Shifting remainders and runends for the blocked RSQF. 23
3.7 Blocked runend-finding algorithm for RSQF. 23
3.8 Performs the blocked equivalent of Equation 3.4, with some optimizations to exit early. 24
3.9 The blocked RSQF's query algorithm. 25
3.10 The blocked RSQF's insertion algorithm. 25
4.1 Encoding a message m. 28
4.2 Approximately multiplying x by 0.782 using bit shifts and additions. 31
5.1 Encoding a message m with arithmetic coding. 38
5.2 Encoding a 64-element array of extensions with arithmetic coding. 38
5.3 Insertion algorithm for the EXTQF. 42
5.4 Query algorithm for the EXTQF. 43
5.5 Adapt over a run in the EXTQF. 44
5.6 Adapt on a single location in the EXTQF. ε is the empty extension. 44
5.7 Insertion algorithm for the SELQF. 47
5.8 Query algorithm for the SELQF. 48
5.9 Adapt over a run in the SELQF. 49
5.10 Adapt on a single location in the SELQF. 49

Abstract

A *filter* is a probabilistic data structure used to represent a set compactly. Unlike a conventional set representation, a filter is allowed to occasionally make mistakes when answering membership queries. This lenient notion of correctness goes hand-in-hand with the filter's compactness. A conventional set can only get so small; once it reaches the point where it cannot get smaller without dropping information, it can get no smaller. A filter, on the other hand, is allowed to pass this boundary. This allowance is precisely what makes the filter small and probabilistic.

Because they are smaller than conventional sets, filters often live higher in the memory hierarchy. As a result, querying a filter takes less time than querying a set. This makes filters quite useful for "filtering out" unnecessary but expensive queries to a set. (Hence the name, "filter".) One can simply query a filter first, using the filter's output to inform whether the set should be consulted.

Classical filters (filters as thus described) are incapable of fixing their mistakes. If a classical filter errs once on an input, it will continue to err on the same input. Consequently, an input stream containing many repeated elements will cause a classical filter to err often. *Adaptive filters*, on the other hand, can fix, or "adapt" to, their mistakes. Thus, they make fewer mistakes overall when confronted with repetitive distributions. Different adaptive filters have different ways of fixing errors, but they all store some data specifically for adaptivity. We call such data *adaptivity bits*.

Filters are often used when building systems, so it is important that they are performant. Prior work on adaptive filters, however, has primarily focused on theoretical designs. Furthermore, the notion of *provable adaptivity* is relatively new, and many theoretical adaptive filters are only heuristically adaptive. Adaptive filters proposed in past work store a fixed amount of information for each element stored in the filter. Recent work has shown, however, that adaptive filters should store variable-length adaptivity bits for each stored element.

This thesis integrates past work on performant static filters and theoretical adaptive filters by developing a technique to make static filters provably adaptive while maintaining good performance. Our technique uses arithmetic coding, a form of data compression, to succinctly represent adaptivity bits. This method allows the use of fractional adaptivity bits for stored elements by grouping elements into blocks and distributing adaptivity bits over blocks.

We demonstrate this technique by developing two practical adaptive filter architectures that are performant and provably adaptive. Theoretical analysis shows that these filters are provably adaptive, and experimental results show that they attain respectable throughputs. While our presentation of the technique is closely tied to a particular implementation, our method for augmenting existing filters with adaptivity is broadly applicable.

Acknowledgments

I worked on the yearlong project that forms the basis of this thesis with Sam and Shikha. They have been wonderful to work with, and I'd like to thank them for their guidance in this project, especially in light of the numerous difficulties of the past year. Special thanks to them both for calling at irregular hours to accommodate time differences, for agreeing to work with me last year without any prior introduction, and for guiding and encouraging me through graduate admissions. Working on filter research with them has been a joy.

I'd also like to thank Shikha again for being a patient thesis advisor. I'm sure my "flexibility" with deadlines annoyed her to no end, but she was kind enough to continue giving me second chances. This thesis would not have been possible without her guidance. Thanks also to Bill Jannen, the second reader of this thesis, for his careful reading and helpful feedback.

Max Stein '21 was also a key contributor to this project. Max worked with us in the summer of 2020 and winter of 2021 and made many contributions to the early design and implementation of our filters. He was great to work with. Over both academic breaks, while we worked remotely from different time zones, I would go to bed after a day of work with the confidence that Max would continue moving forward while I slept.

Warm thanks are also due to Stephen Freund for introducing me to computer science research in the summer of 2019 and for his guidance through graduate admissions.

Finally, I'd like to thank my parents, Chang Kyoo Lee and Young Sil Kim, my sister, Jennifer, and my girlfriend, Maki, for their unwavering support. Living through the pandemic has been terrible in many regards, but spending time with family while working remotely was a delight. And whether in person or over the internet, Maki has been a great emotional support.

Chapter 1

Introduction

A *filter* is a data structure that compactly and approximately represents a set. Unlike a conventional set representation, a filter is allowed to err. This allows a filter to trade correctness in exchange for size. Although a filter is allowed to make mistakes, its mistakes are constrained by two guarantees. First, a filter's error is *bounded*. This comes in the form of an error rate parameter ε. Lower error rates yield larger filters; higher error rates enable smaller filters. Second, a filter's error is *one-sided*. A filter is only allowed to report false positives (the filter can report that an element is in a set when it is not) and not false negatives (the filter cannot report that an element is not in a set when it is).

Filters have a variety of applications. Because they can represent sets more compactly than conventional set representations, which cannot err, filters can sit higher in the memory hierarchy. This makes filters particularly useful for weeding out unnecessary but expensive queries to a larger data store. Modern filter applications include databases, LSM-based key-value stores [7, 14], and distributed systems [6].

Classical filters do not fix their mistakes. Absent an operation that unintentionally changes internal state, a query that is once a false positive will stay a false positive. This means that a filter's guaranteed error rate ε only holds for individual queries and not for sequences of queries. Thus, classical filters may perform poorly on data that contains repetitions—for example, data that follows a power-law distribution, or a sequence of queries fed into a filter by an adversary.

Adaptive filters fix their mistakes; as a result, they make fewer mistakes overall when confronted with repetitive or adversarial distributions. Heuristically adaptive filters were first introduced by Mitzenmacher [12], who showed that a heuristically adaptive cuckoo filter could outperform a *static* (i.e., non-adaptive) filter on repetitive data. This work was followed up by Bender et al. [2], who introduced the theoretical notion of *provable adaptivity* based on an adversarial game. Work by Kopelowitz et al. [10] corroborated Mitzenmacher's work on the performance benefits of adaptive filters but showed that, even for a weaker, non-adversarial notion of adaptivity, Mitzenmacher's heuristically adaptive cuckoo filters fail to be adaptive. Kopelowitz et al. also showed that a prerequisite to achieving adaptivity is allocating a *variable* number of bits to each stored element. Notably, all past work on adaptive filters has used modified cuckoo filters that allocate a *constant* number of bits per element for adaptivity [10, 12].

Bender et al.'s work on provable adaptivity [2] included a theoretical filter called the *broom filter*, which was named as a play on the well-known Bloom filter [5]. The broom filter is provably adaptive but lacks a fast practical implementation. More recent work on the broom filter [4] uses an implementation that, the authors note, is "quite slow".

Because filters are often used in a system-building context, it is important that they are performant in practice. The primary challenge of designing a practical adaptive filter is that, in a filter, *every bit counts*. Storing a single extra bit per stored element can make a significant difference in filter performance. Accordingly, budgeting a single bit per element for adaptivity has a steep opportunity cost. The challenge of designing a performant adaptive filter is precisely the challenge of developing an adaptivity mechanism that is effective enough to justify every bit it requires. Storing a variable number of bits per element in a succinct and reconstructible way further complicates this task.

To address the gap in the literature between static filters that are useful in practice and theoretically adaptive filters that are challenging to implement, we introduce two practical adaptive filter implementations: the extension-based adaptive quotient filter (ExtQF) and the selector-based adaptive quotient filter (SelQF). The ExtQF is a practical implementation of the broom filter, and uses *fingerprint extensions* to resolve false positives. The SelQF uses *hash selectors*, first introduced by Mitzenmacher et al. [12], to achieve the same effect. The construction and empirical evaluation of these filters form the primary contributions of this thesis.

The SelQF and ExtQF are based on a shared core architecture. This architecture begins with the rank-and-select quotient filter (RSQF), developed by Pandey et al. as the basis of the counting quotient filter [16]. Atop the RSQF, we add adaptivity information using *arithmetic coding*, a compression technique. Arithmetic coding offers compression that is very close to optimal in practice; allows the use of fractional bits per encoded symbol on average; and performs well on an exponentially decaying probability distribution. All three properties are important to satisfy. Optimally succinct encodings minimize adaptivity's footprint; fractional encodings enable the allocation of adaptivity bits to the elements that need them most; and the relevant probabilities exhibit exponential decay. Arithmetic coding's primary downside is that it is slow [9], but we introduce an efficient variant of arithmetic coding that resolves this issue.

Experimental results show that the SelQF outperforms the ExtQF significantly on false positive performance, indicating that hash selectors use adaptivity bits more efficiently than fingerprint extensions. The SelQF itself matches or outperforms other heuristically adaptive filters. Both the SelQF and ExtQF achieve comparable throughput to their non-adaptive counterparts.

While this thesis focuses on the presentation of the ExtQF and SelQF, we expect that the techniques developed to make the RSQF adaptive are broadly applicable to other filter architectures.

1.1 Overview

This thesis is presented as follows.

Chapter 2 gives more detailed background on filters and adaptive filters. We motivate filters, discuss their principal architectures, and review recent work on adaptivity.

Chapter 3 details the design and implementation of the rank-and-select quotient filter (RSQF), whose structure undergirds our two adaptive filters. We define RANK and SELECT, introduce a simplified RSQF to show how RANK and SELECT may be used in a quotient filter, and finally introduce the RSQF itself as a refinement of the simplified RSQF.

Chapter 4 introduces arithmetic coding, a compression technique that represents messages as points in an interval. Arithmetic coding has many desirable properties, but its naïve implementation is quite slow. We discuss optimizations that make arithmetic coding performant enough to be used in practice.

Chapter 5 describes our practical adaptive quotient filters. We discuss the broom filter in more detail, enumerate the key components of an adaptive filter, and present the designs of the SELQF and EXTQF.

Chapter 6 presents experimental results. We evaluate the accuracy and throughput of our filters on various data sets and compare our results with those of other state-of-the-art filters.

Chapter 7 reviews our contributions and considers directions for future work.

Chapter 2

Background

A filter is a succinct approximate set membership data structure. A filter's succinctness and approximate correctness go hand-in-hand: because of its relaxed correctness guarantees, a filter can be smaller than a perfect set membership data structure. Filters are often used in key-value stores [7, 14] and network applications [6]. More generally, filters are useful in applications where accesses to conventional sets are expensive, or where throughput is paramount and small amounts of error are tolerable.

The Bloom filter [5] is a classic, well-known example. More recent work has developed the cuckoo [8] and quotient [3] filter architectures. We discuss each architecture in turn. Following this discussion, we define adaptivity [2] for filters, develop the requirements for building an adaptive filter, and summarize past work.

2.1 Filters

In the simplest terms, a *filter* is a compact and lossy set representation. Because it is allowed to err, a filter can use fewer bits than a set representation that is not allowed to err. The goal of a filter is to use minimal space to approximately answer set membership queries. In other words, for a filter Q representing the set S, Q answers queries of the form, "Is x in S?" For this reason, filters primarily support just two operations[1]—insertions and queries—and are also referred to as *approximate membership query* (AMQ) data structures.

A filter's error is constrained by two guarantees. First, the rate at which a filter reports erroneous results is bounded. Each filter has a tunable error probability ε that holds on individual queries. Second, the error is one-sided: a filter is allowed to report false positives but not false negatives. That is, for $x \in S$, the filter will never report "$x \notin S$" (ABSENT), but for $x \notin S$, the filter may report "$x \in S$" (PRESENT), with the probability of this occurrence being bounded by an error rate ε. Stated another way, if the filter reports "$x \notin S$", it is certainly correct; if it reports "$x \in S$", it is probably correct.

Because a filter representing S is smaller than an exact representation of S, it often lives higher in the memory hierarchy than S and benefits from faster accesses. This observation makes filters

[1] Some filters support deletions, but this often comes at the expense of simplicity or performance.

valuable in a system-building context, where they are used to speed up slow, expensive queries to large key-value stores. Instead of looking up a key directly in the database, we can query a filter first, consulting the database only when we are confident that the key is present. With the database serving as the ground truth, the filter can skip some queries without introducing correctness issues.

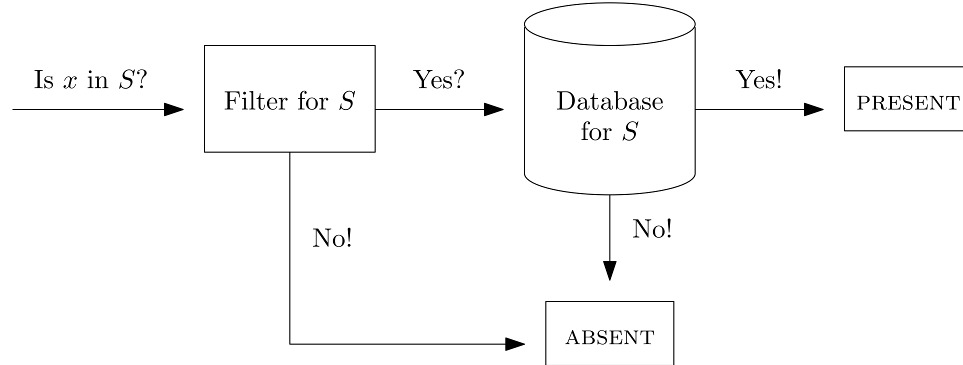


Figure 2.1: Filters can cut out unnecessary but expensive queries early with high probability. Reproduced from [17].

Our earlier stipulation that error must be one-sided serves us well in this context. If both positives ($x \in S$) and negatives ($x \notin S$) were uncertain, all queries would have to consult the database. But because a filter's error is one-sided, we must consult the database only when the filter's result is uncertain.

When a filter reports ABSENT, we can trust it, as filters report no false negatives. This lets us avoid an expensive query, saving time. When a filter reports PRESENT, on the other hand, we must check the database to be sure. In the case of a false positive, this results in an unnecessary access. Without the filter, however, this access would have occurred anyway—it is not a new cost. The added cost of querying the filter is relatively insubstantial, owing to the filter's small size. Therefore, even in the case of a false positive, we add negligible cost by consulting the filter before checking the database.

On balance, the time savings from cutting out unnecessary negative queries (with high probability) more than make up for the cost of consulting the filter. Furthermore, because oftentimes a set is small relative to the universe from which its elements are drawn, queries skew negative, further tipping the balance in favor of filters.

When designing a filter, the desired qualities are compactness, speed, and accuracy. A smaller filter fits in smaller storage media higher up in the memory hierarchy and therefore has shorter access times. But to take full advantage of these short access times, the filter's intrinsic operations (insertions and queries) must also be fast. Moreover, for the filter to effectively filter queries, it must maintain a low false positive rate: a fast, tiny filter with a high false positive rate may be less useful than a slower, larger filter that answers queries with fewer false positives.

These three qualities are often in opposition: improving a filter in one dimension often entails a sacrifice in another dimension. For example, storing extra information can improve a filter's speed or accuracy, but it also increases the filter's size. In general, the filter designer's task is to work within

a space budget to make a fast and accurate filter. Accordingly, comparing different filter designs involves giving each filter the same amount of space and seeing how fast or accurate the filters are in relation to one another.

Our discussion thus far has developed a specification for the filter data structure. In the following sections, we discuss specific filter architectures that meet this specification. Filters have three principal architectures—Bloom, cuckoo, and quotient.

2.1.1 Bloom filter

The *Bloom filter* was the first filter, and was developed by Burton Bloom in 1970 [5]. Thanks to its relative simplicity, the Bloom filter provides a good introduction to filter implementation and theoretical analysis.

A Bloom filter for a set S of n elements drawn from a universe U consists of a bitarray B of m bits and a collection of k hash functions $h_1, h_2, ..., h_k$, where each $h_i : U \to [0, m-1]$. An empty Bloom filter has all of the bits in its bitarray set to 0. Recall from earlier that filters support two operations: insertions and queries.

Insertions

To add an element $x \in S$ to a Bloom filter, we hash x using each of the k hash functions, yielding $h_1(x), h_2(x), ..., h_k(x)$. Then, we ensure that all of the corresponding bits in the bitarray B are set to 1. That is, we set bits in B such that $B[h_i(x)] = 1$ for $i \in \{1, ..., k\}$.

Queries

To ask the filter whether an element $y \in U$ is in S, we hash y with the k hash functions and check whether all of the corresponding bits in the bitarray are 1. In other words, we check whether $B[h_i(y)] = 1$ for $i \in \{1, ..., k\}$. If all of the appropriate bits are set, then the Bloom filter reports PRESENT; if any are unset, then the filter reports ABSENT.

Analysis

By design, the Bloom filter will report no false negatives. A false negative occurs when the filter reports ABSENT for some $x \in S$. But if $x \in S$, x must have been inserted into the filter, so all of its corresponding bits will have been set. Therefore, querying x will lead the filter to always report PRESENT. On the other hand, the filter may report false positives: the corresponding bits for x, $h_1(x), ..., h_k(x)$, may have been set by other elements in S whose hashes collide with those of x.

The size of the set, $n = |S|$, and the desired false positive rate ε are often decided by the application context. Given n and subject to the constraint ε, we can minimize m and choose an optimal value for k. Indeed, to ensure that the expected false positive rate is bounded by ε, we require $k = \ln 2 \cdot m/n$ and $m \geq n \log e \cdot \log(1/\varepsilon)$ [17, 6]. The minimal amount of space required to represent S with false positive rate ε is approximately $n \log(1/\varepsilon)$, so Bloom filters are within approximately $\log e \approx 1.44$ of the lower space bound. Lookups require $k = \ln 2 \cdot (m/n) = \log(1/\varepsilon)$ hashes and reads; insertions require k hashes and writes.

2.1.2 Cuckoo filter

A *cuckoo filter* compactly represents a set by storing hashes of set elements instead of the elements themselves. These hashes are called *fingerprints*. Cuckoo filters get their name from the fact that they resolve hash collisions using *cuckoo hashing* [8, 12], which in turn derives its name from the brood parasitism of some species of cuckoo. Cuckoos lay their eggs in the nests of other birds; upon hatching, cuckoo chicks evict other eggs from the nest. Cuckoo hashing is an analogous policy for handling hash collisions. When a new key is inserted into a location that already holds an older key, the older key is evicted and placed into a different slot.

A cuckoo filter for a set $S \subseteq U$ of n elements consists of k hash functions $h_1, h_2, ..., h_k$ where each $h_i : U \to [0, m-1]$, a fingerprint hash function $f : U \to [0, \frac{1}{\varepsilon} - 1]$, and a hash table T of m buckets, where each bucket can store c fingerprints. For simplicity, assume $c = 1$. In an empty cuckoo filter, all slots in T are marked as empty. This can be done explicitly by using an extra bit to denote whether a slot is empty or full, or implicitly by narrowing the range of f to $[1, \frac{1}{\varepsilon} - 1]$ and using 0 to mark slots as empty.

Insertions

To insert an element $x \in S$ into a cuckoo filter, we first check if $T[h_1(x)]$ is empty. If it is, we store $f(x)$ in $T[h_1(x)]$. If it is not, we check $T[h_2(x)], ..., T[h_k(x)]$ in order. We store $f(x)$ in the first empty $T[h_i(x)]$.

If, for all $i \in \{1, ..., k\}$, $T[h_i(x)]$ is nonempty, then we choose some $i \in \{1, ..., k\}$, and let $f(y)$ be the fingerprint stored at $T[h_i(x)]$. We store $f(x)$ in $T[h_i(x)]$ and "cuckoo" $f(y)$ to another slot. If there exists some $j \in \{1, ..., k\} \setminus \{i\}$ such that $T[h_j(y)]$ is empty, then we store $f(y)$ in $T[h_j(y)]$. Otherwise, we pick some $j \in \{1, ..., k\} \setminus \{i\}$ and store $f(y)$ in $T[h_j(y)]$, cuckooing the original occupant $f(z)$ to another slot. We continue until either all elements are placed into new slots or $\log n$ elements are cuckooed. In the latter case, we rebuild the filter.

The careful reader will have noticed that in the above algorithm, we obtain $h_j(y)$ from $f(y)$ and $h_j(y)$ without knowing the value of y. We can resolve this issue for any k by using an additional data structure to map $f(y)$ back to y; for example, with a key-value store. When $k = 2$, however, we can resolve the issue without storing additional data using *partial-key cuckoo hashing*. That is, we keep $h_1(x)$ as is, but define another hash function h and let $h_2(x) = h_1(x) \oplus h(f(x))$, making sure that $h(f(x))$ is nonzero.[2] Because XOR is associative and is its own inverse, it follows that

$$\begin{aligned} h_1(x) &= h_1(x) \oplus 0 \\ &= h_1(x) \oplus \Big(h(f(x)) \oplus h(f(x))\Big) && \text{inverse} \\ &= \Big(h_1(x) \oplus h(f(x))\Big) \oplus h(f(x)) && \text{associativity} \\ &= h_2(x) \oplus h(f(x)). \end{aligned}$$

Thus, to cuckoo $f(y)$ from slot $h_1(y)$, we can find the next slot $h_2(y)$ from $h(f(y))$ and $h_1(y)$.

[2] Otherwise, $h_2(x) = h_1(x) \oplus 0 = h_1(x)$, and x will have only one slot it can hash to.

Queries

To query a cuckoo filter about the membership of an element $x \in U$ in S, we check each of the slots $T[h_1(x)], T[h_2(x)], ..., T[h_k(x)]$. If there exists a slot for which $T[h_i(x)] = f(x)$, then the filter returns PRESENT. Otherwise, the filter returns ABSENT.

Analysis

First, we check that the cuckoo filter allows no false negatives. For each element inserted into the filter, its fingerprint resides in at least one of the slots corresponding to the element's k hashes. In other words, for an element $x \in S$, there exists some $i \in \{1, ..., k\}$ for which $T[h_i(x)] = f(x)$. This guarantees that for any $x \in S$, the cuckoo filter will return PRESENT.

Next, we check that the cuckoo filter's false positive rate is bounded by ε. A query $x \notin S$ is a false positive if the filter returns PRESENT, which happens when there exists some $i \in \{1, ..., k\}$ such that $T[h_i(x)] = f(x)$. To compute the probability of this event, we first consider the probability that a slot $T[h_j(x)]$ stores a fingerprint matching $f(x)$.

Let $\Pr(A)$ be the probability that slot $T[h_j(x)]$ is nonempty, and let $\Pr(B)$ be the probability that the fingerprint $f(y)$ stored in slot $T[h_j(x)]$ matches $f(x)$. Because the cuckoo filter stores n elements in m slots, $\Pr(A) = n/m$. Given that $T[h_j(x)]$ is nonempty, the probability that the fingerprint in slot $T[h_j(x)]$ matches $f(x)$ is $\Pr(B|A) = 1/(1/\varepsilon) = \varepsilon$. Therefore, the probability that slot $T[h_j(x)]$ stores a fingerprint matching $f(x)$ is $\Pr(A \cap B) = \Pr(A)\Pr(B|A) = n/m \cdot \varepsilon$.

This analysis holds for any $j \in \{1, ..., k\}$, so the probability of a false positive is bounded by $\sum_{i=1}^{k} n/m \cdot \varepsilon$ by the union bound. When $m = kn$, this bound is equal to ε, so the false positive probability is bounded by ε.

When $k = 2$, the cuckoo filter uses $2n \log(1/\varepsilon)$ bits, as compared to the Bloom filter's $1.44 \log(1/\varepsilon)$ bits. We can obtain better space utilization while maintaining good performance by setting $k = 4$, using a hash table T with only $1.05n$ slots, and fingerprints of length $\log(4/\varepsilon)$ bits [8]. This uses $1.05n \log(4/\varepsilon) = 1.05n \log(1/\varepsilon) + 2n$ bits. Thus, cuckoo filters are more space efficient than Bloom filters when $\varepsilon < 3\%$.

2.1.3 Quotient filter

The *quotient filter* is a practical implementation of the *single hash function filter*, introduced by Pagh et al. [15]. Like cuckoo filters, quotient filters represent sets compactly by storing fingerprints. Unlike cuckoo filters, quotient filters store fingerprints via a technique called *quotienting*, from which quotient filters derive their name. Quotienting involves splitting the p-bit fingerprint $f(x)$ of an element x into two parts: a q-bit *quotient* and an r-bit *remainder*, where $p = q + r$, $q = \log n$, and $r = \log 1/\varepsilon$. We denote the quotient of x by $\mathrm{quot}(x)$ and the remainder of x by $\mathrm{rem}(x)$.

The key idea behind quotienting is to save space by storing only remainders, using the positions where remainders are stored to infer their associated quotients. In effect, a quotient filter is a table T of n r-bit remainders, supplemented with metadata bits for bookkeeping, where n is the size of the stored set. Roughly speaking, an element x is represented in a quotient filter by storing $\mathrm{rem}(x)$ at $T[\mathrm{quot}(x)]$.

This is a useful big-picture view, but it is clearly problematic under further examination. For

example, consider elements $x, y \in S$ where $\text{quot}(x) = \text{quot}(y)$ and $\text{rem}(x) < \text{rem}(y)$. Both x and y, it would seem, belong in slot $T[\text{quot}(x)]$. Quotient filters handle this case by using a kind of linear probing: fingerprints are kept ordered with respect to their quotients, and overlaps are handled by shifting over elements with higher quotients into the next available slot—in this case, placing $\text{rem}(x)$ in $T[\text{quot}(x)]$ and $\text{rem}(y)$ in $T[\text{quot}(x)+1]$.

More generally, when multiple fingerprints share the same quotient, a quotient filter stores quotient-sharing remainders in a contiguous sequence called a *run*. When two runs overlap, the run with the lesser quotient pushes any subsequent runs to later slots. Concretely, when the run for a quotient q_1 overlaps with the run for a quotient q_2 where $q_1 < q_2$, the quotient filter will shift the run for q_2 such that the first remainder in q_2's run will immediately follow the last remainder in q_1's run (granted that there are no additional runs in between).

As a result, a quotient filter's elements satisfy a few useful invariants [16]. First, for all $x, y \in S$, if $\text{quot}(x) < \text{quot}(y)$ then $\text{rem}(x)$ is stored in an earlier slot than $\text{rem}(y)$. Second, the remainder $\text{rem}(x)$ of an element $x \in S$ is stored at a slot in T at least $\text{quot}(x)$. Third, if $\text{rem}(x)$ is stored at an index $i > \text{quot}(x)$, then the slots in $T[\text{quot}, ..., i-1]$ are all occupied by other remainders whose quotients are at most $\text{quot}(x)$.

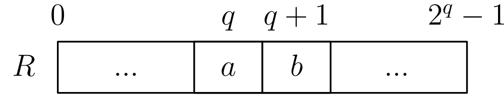


Figure 2.2: Without metadata, a table of remainders can be interpreted in multiple conflicting ways.

The problem remains that storing a table of remainders alone, even with a policy for managing shared quotients, is insufficient. Without additional information, a table of remainders can be interpreted in multiple different ways. For example, the table T containing remainders a and b at quotients q and $q+1$, respectively (Figure 2.2), could be interpreted as any one of the following sets of quotient-remainder pairs:

$$
\begin{array}{llll}
\{(q,a),(q,b)\} & \{(q-1,a),(q-1,b)\} & ... & \{(0,a),(0,b)\} \\
\{(q,a),(q+1,b)\} & \{(q-1,a),(q,b)\} & ... & \{(0,a),(1,b)\} \\
 & \{(q-1,a),(q+1,b)\} & ... & \{(0,a),(2,b)\} \\
 & & & ... \\
 & & & \{(0,a),(q+1,b)\}
\end{array}
$$

The remainder a could belong to the run of any quotient $\leq q$; similarly, b could belong to the run of any quotient $\leq q+1$.

To determine which remainders are associated with which quotients, quotient filters store additional metadata bits from which this information may be inferred. The original quotient filter [3] stores 3 metadata bits per element. The rank-and-select quotient filter (RSQF) [16], on which we base the bulk of our work, uses 2.125 metadata bits per element. We describe the quotient filter's

insertion and query algorithms at a high level here and defer the details to Chapter 3.

Insertions

To insert an element x into a quotient filter, we use the filter's associated metadata to find the run associated with $\text{quot}(x)$. If such a run exists, then we extend the existing run: we insert $\text{rem}(x)$ after the last element in the run and shift over any remainders that get in the way, taking care to preserve the quotient filter's invariants. If no such run exists, then we start a new run. If $T[\text{quot}(x)]$ is empty, we start the new single-element run here; if it is not, then we start the run after $\text{quot}(y)$'s run, where $\text{quot}(y)$ is the largest quotient with a run in the filter such that $\text{quot}(y) < \text{quot}(x)$.

Queries

To check whether $x \in S$, we use the filter's associated metadata to find $\text{quot}(x)$'s run and scan through it to find $\text{rem}(x)$. If the run doesn't exist, then the filter returns ABSENT. If the run exists and there exists a remainder $\text{rem}(y)$ in the run such that $\text{rem}(y) = \text{rem}(x)$, then the filter reports PRESENT; otherwise, it reports ABSENT.

Analysis

First, we check that a quotient filter allows no false negatives. If $x \in S$, x is inserted into the quotient filter, so $\text{rem}(x)$ is in the run for $\text{quot}(x)$, and the filter will report PRESENT.

Next, we consider the quotient filter's false positive rate. In the quotient filter, false positives occur when an element in the set has a hash collision with an element not in the set. Formally, a false positive arises when $\exists x \in S,\ y \notin S$ such that $f(x) = f(y)$. The probability that $f(x) = f(y)$ is precisely the probability that all q quotient bits and all r remainder bits of $f(x)$ and $f(y)$ are equivalent. Therefore,

$$\begin{aligned}
\Pr(f(x) = f(y)) &= 2^{-(q+r)} \\
&= 2^{-(\log n + \log(1/\varepsilon))} \\
&= 2^{-\log(n/\varepsilon)} \\
&= \frac{1}{n/\varepsilon} \\
&= \varepsilon/n.
\end{aligned}$$

By the union bound, the probability that $f(y) = f(x)$ for any $x \in S$ is upper-bounded by $n \cdot \varepsilon/n = \varepsilon$. Therefore, the quotient filter satisfies the false positive probability bound ε.

A quotient filter uses $n \log(n/\varepsilon)$ bits to store T and, in the case of the original quotient filter, $3n$ metadata bits, for a total of $n \cdot (3 + \log(n/\varepsilon))$ bits. The RSQF uses $2.125n$ metadata bits, for a total of $n \cdot (2.125 + \log(n/\varepsilon))$ bits.

2.2 Adaptivity

The filters we have discussed thus far are all *static* filters: they do not correct their false positives. If querying a static filter with x yields a false positive, then querying the filter with the sequence $x, x, \ldots$ will drive the filter's false positive rate to 1. In other words, a static filter's false positive

probability bound ε applies only to single queries, and does not extend to sequences of queries. Static filters will tend to yield elevated false positive rates on repetitive data, whether the repetition is intentional—and malicious—or naturally occurring.

In contrast to static filters, *adaptive* filters fix past false positives, guaranteeing that their false positive rates apply to bounded[3] sequences of queries. Adaptive filters are well suited for Zipfian or adversarial distributions, as they can fix the repeated false positives that trouble classical filters.

2.2.1 Remote representation

For a filter to fix false positives, it must be able to identify false positives when they occur. When a filter is used to speed up queries to a dictionary (Figure 2.1), it is easy to identify false positives using the dictionary and serve this information to the filter. Sending this feedback to the filter is essentially free, because it requires no additional accesses to the dictionary.

A result by Bender et al. [2] shows that even with feedback from an external dictionary about whether a query x is a false positive, a filter must use at least $\Omega(n \log\log u, n \log n)$ bits to be adaptive, where $n = |S|$ and $u = |U|$. In other words, an adaptive filter needs to know *which element* $y \in S$ is causing a query element x to be a false positive, not merely *whether* x is a false positive.

To prevent the increased size requirement from affecting the filter's performance, adaptive filters partition their state into two pieces: a small *local representation* L and a larger *remote representation* R. L is a static filter augmented by $O(n)$ bits for adaptivity; R is an oracle that informs L of which stored element caused a false positive, and is only accessed when the external dictionary is accessed to avoid slowing down operations on L.

2.2.2 Prior work on adaptive filters

Prior work on adaptive filters has focused on theoretical designs that modify the three primary filter architectures to make them adaptive. We summarize important recent work.

Adaptive Cuckoo Filter

In 2017, Mitzenmacher [12] proposed the *adaptive cuckoo filter* (ACF). The ACF pairs a cuckoo filter with a cuckoo hash table that is updated in lock-step with the filter. The hash table and filter behave identically except that the former stores set members in full and the latter stores fingerprints. In effect, the table acts as a remote representation that helps the filter (the local representation) adapt. In the filter, each fingerprint is stored with an additional *hash selector* that denotes the index i of the fingerprint hash function f_i used to generate the fingerprint. Each hash selector uses s bits. Equivalently, the ACF maintains $2^s - 1$ fingerprint hash functions $f_0, f_1, ..., f_{2^s-1}$.

When the filter reports a positive for a query on an element y with hash selector a, i.e., $f_a(y) = f_a(x)$ for some $x \in S$, the ACF determines whether it is a false positive by checking whether $x = y$ in the hash table. It then fixes false positives by re-hashing x using the k-th fingerprint hash function f_k, where $k = (a + 1) \bmod (2^s - 1)$. Mitzenmacher simulates the ACF to study its performance on real-world packet trace data [18], which exhibits repetitions, and finds that the ACF outperforms the static cuckoo filter.

[3] Given a long enough sequence of queries, any filter with finite memory will fail to fix past false positives.

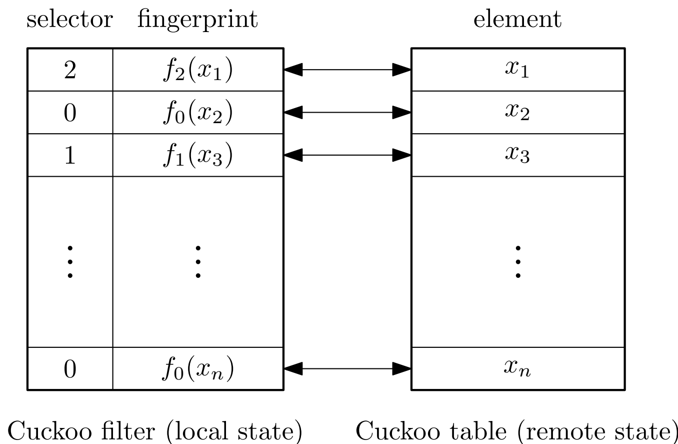


Figure 2.3: A simplified view of the Adaptive Cuckoo Filter. Elements in the remote state are updated in sync with fingerprints in the local state

In more recent work, Kopelowitz et al. [10] show that Mitzenmacher's heuristically adaptive ACFs are not adaptive by Bender et al.'s adversarial notion of adaptivity [2], nor are they adaptive by a weaker, non-adversarial notion of adaptivity [10]. They find that a prerequisite to achieving adaptivity is allocating a variable number of bits to each stored element.

Broom Filter

In 2018, Bender et al. [2] formalized the notion of adaptivity and established lower bounds for the size of an adaptive filter. They showed that adaptive filters require large space and resolved this issue by partitioning adaptive filters into a small local state and a larger remote state, as discussed previously.

In the same work, Bender et al. introduced the *broom filter* (named as a play on the Bloom filter), an augmented quotient filter that fixes false positives by extending the remainders of the elements at fault. That is, when a false positive occurs for $x \in S$ and $y \notin S$ because $f(x) = f(y)$, the broom filter changes its internal state, storing enough extra bits of $\text{rem}(x)$ to disambiguate $f(x)$ from $f(y)$. These extra bits are called *adaptivity bits*. To ensure that there are enough bits in the fingerprint function to allow adding bits as needed, the broom filter requires a fingerprint function with sufficiently many bits: $f : U \to n^c$ for some constant $c \geq 4$.

In a more recent paper [4], Bender et al. compare the broom filter to a static filter augmented with a comparably sized TOP-k cache. (A TOP-k cache holds onto the k requests that have the highest observed frequencies.) They find that the broom filter outperforms the cache-augmented filter on Zipfian distributions thanks to "serendipitous corrections". By extending a remainder by one bit, the broom filter eliminates future false positives in addition to the false positive that triggered the extension. Interestingly, they find that serendipitous corrections account for 24-55% of all of the false positives avoided on network trace data.

They note, however, that their broom filter implementation is "quite slow." Perhaps for this reason, their experiments are conducted on a tiny broom filter that stores only 30 elements. In

Chapter 5.3, we design and implement a fast version of the broom filter.

Cuckooing Filter

In past work, Kopelowitz et al. [10] developed the *cuckooing adaptive cuckoo filter*, which adapts on false positives by cuckooing the elements that triggered them. That is, when the cuckooing filter returns a false positive for some $x \notin S$, the filter cuckoos any elements that collide with x as it would during an insertion. This modifies the filter's internal state so that future queries on x will return "$x \notin S$".

2.3 Summary

Filters are compact probabilitic set representations with bounded and one-sided error. At present, they are primarily used to speed up queries to large and slow databases. There are three main filter architectures, each with their own ideas of concisely representing a lossy set.

Adaptive filters generalize classical filters' error guarantees from single queries to sequences of queries. Viewed another way, adaptive filters fix the false positives they encounter. To detect false positives when they occur, adaptive filters employ a remote representation—a large perfect set representation. Past work on adaptive filters has led to the development of the broom filter, various adaptive cuckoo filters, and to formal notions of adaptivity.

Chapter 3

Rank-and-Select Quotient Filter

This work builds upon the *rank-and-select quotient filter* (RSQF) [16], a performant quotient filter that achieves higher throughput at a smaller size than the original quotient filter. As such, it is important to understand the RSQF—its structure, strengths, and weaknesses—to understand how it may be made adaptive.

In this chapter, we introduce the RSQF. We first define the RANK and SELECT operations and show how they enable the RSQF to find and manage runs efficiently. We begin this discussion by considering insertion and query algorithms for a simplifed, single-array RSQF. After these preliminaries, we describe the blocked RSQF architecture, which segments data into blocks to optimize cache performance.

3.1 Rank and select

The RSQF takes its name from the RANK and SELECT operations, which are often used in succint data structures. We define these operations and show how they may be implemented efficiently using x86 instructions on 64-bit words.

3.1.1 Definition

Definition 3.1 (RANK)**.** For a bitarray B and index $i \in \{0, 1, .., |B|-1\}$, the RANK operation counts the number of 1 bits in B up to i. Formally,

$$\text{RANK}(B, i) = |\{k \in 0, 1, ..., i \mid B[k] = 1\}|.$$

Definition 3.2 (SELECT)**.** For a bitarray B and rank $k \in \{1, .., |B|\}$, the SELECT operation finds the index of the k-th 1 bit in B. Formally,

$$\text{SELECT}(B, k) = \min\{i \in 0, 1, ..., k-1 \mid \text{RANK}(B, i) = k\}.$$

Instruction	Full name	Arguments	Behavior
`popcnt`	*Pop*ulation *Count*	*dst, src*	`popcnt` counts the number of bits set to 1 in *src*.
`tzcnt`	*T*railing *Z*ero *Count*	*dst, src*	`tzcnt` counts the number of trailing least significant zero bits in *src*. When *src* $= 0$, `tzcnt` returns the size of *src* (64).
`pdep`	*P*arallel Bits *Dep*osit	*dst, src, mask*	`pdep` copies contiguous low order bits from *src* into locations in *dst* corresponding to bits set to 1 in *mask*. All other bits (i.e., 0 bits in *mask*) are set to 0.

Figure 3.1: Haswell instructions used to implement fast RANK and SELECT operations.

3.1.2 Fast implementation

We can implement[1] RANK and SELECT efficiently for 64-bit words using instructions in the x86 instruction set for Haswell processors and newer [16], as shown in Figure 3.1. We implement RANK by masking the first i bits of a bitarray B applying `popcnt` on the result (Equation 3.1). We implement SELECT using `pdep` to deposit a 1-bit at the position of the i-th 1-bit in B and `tzcnt` to determine the location of the deposited bit (Equation 3.2).

$$\textsc{rank}(B, i) = \texttt{popcnt}(B \;\&\; (2^i - 1)) \tag{3.1}$$

$$\textsc{select}(B, i) = \texttt{tzcnt}(\texttt{pdep}(2^i, B)) \tag{3.2}$$

3.2 Single-block RSQF

Equipped with fast RANK and SELECT operations, we now consider a simplified RSQF consisting of a single large array. Like the original quotient filter (QF), the RSQF represents the elements of a set S using a fingerprinting hash function f. A fingerprint $f(x)$ is split into a quotient $\mathrm{quot}(x)$ of $\log n$ bits and a remainder $\mathrm{rem}(x)$ of $\log(1/\varepsilon)$ bits.

The quotient is stored implicitly and the remainder is stored explicitly. Remainders are stored in an array of n slots, and their placement, along with some metadata bits, can be used to reconstruct the full fingerprint that a remainder was taken from. Remainders that share the same quotient are stored in contiguous sequences called *runs*. In summary, the RSQF, like the QF, marks the presence of a fingerprint $f(x) = \mathrm{quot}(x) \circ \mathrm{rem}(x)$ by storing $\mathrm{rem}(x)$ in $\mathrm{quot}(x)$'s run.

The RSQF's primary contribution is the clever use of RANK and SELECT operations to quickly find a particular quotient's run. The RSQF also requires fewer metadata bits per element than the

[1] In Equations 3.1 and 3.2, we omit the first operand (*dst*) in each instruction, treating the *dst* operand as the value of the instruction applied on its other operands. For example, we express *dst* in `popcnt(dst, src, mask)` as `popcnt(src, mask)`.

QF does: 2.125 metadata bits per element to the QF's 3.

The RSQF stores two metadata bitarrays of n bits. The first bitarray, θ, stores *occupied* bits; the second, ρ, stores *runend* bits. The occupied bit at i indicates whether any elements with the quotient i have been inserted into the filter. The runend bit at i tracks whether the remainder placed at slot i is the last remainder in a run of quotient-sharing remainders.

It is helpful to consider some examples. An RSQF containing a single element with the fingerprint $q \circ r$ is shown in Figure 3.2. Both $\theta(q)$ and $\rho(q)$ are set to 1; all other metadata bits are set to 0; and the remainder at q is set to r. In the next example (Figure 3.3), the RSQF contains the three quotient-remainder pairs $(q, r_1), (q, r_2)$, and $(q + 1, r_3)$. Note that the third pair has quotient $q + 1$ but is placed in slot $q + 2$ because of the second pair.

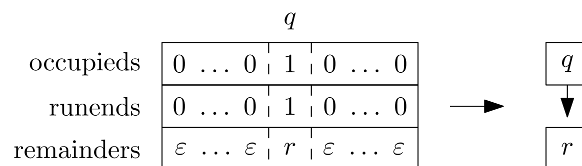


Figure 3.2: A singleton run and its interpretation.

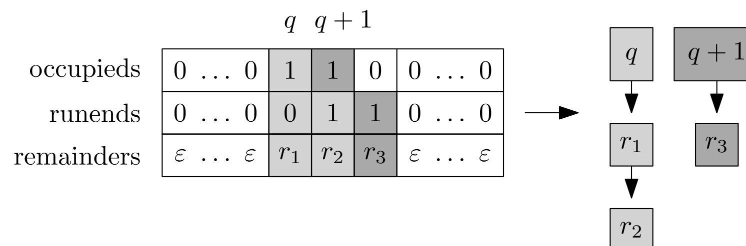


Figure 3.3: Multiple runs and their interpretations.

In summary, we define the simplified RSQF Q as the triple (θ, ρ, σ), where θ is an n-bit bitarray storing occupied bits; ρ is an n-bit bitarray storing runend bits; and σ is an n-element array of $log(1/\varepsilon)$-bit remainders.

Symbol	Meaning
θ	bitarray of occupied bits
ρ	bitarray of runend bits
σ	array of remainders

Figure 3.4: RSQF notation.

3.2.1 Membership queries

At a high level, membership queries are identical to the QF's. Checking the presence of a fingerprint $q \circ r$ entails traversing the run associated with q to find r. This can be broken down into two steps:

1. Find the end of the run associated with q.

2. Walk backwards from the end of the run until we find r or reach the beginning of the run.

The two metadata bitarrays θ and ρ enable us to quickly find the end of a quotient's run. Consider a particular element $x \in S$, where the RSQF $Q = (\theta, \rho, \sigma)$ represents the set S. If the quotient's occupied bit is unset ($\theta[q] = 0$), then the RSQF immediately reports ABSENT. If the quotient's occupied bit is set ($\theta[q] = 1$), then the RSQF finds the end of q's run by using RANK and SELECT:

$$m = \text{RANK}(\theta, q) \tag{3.3}$$

$$s = \text{SELECT}(\rho, m). \tag{3.4}$$

Equation 3.3 uses RANK to determine the number of quotients in Q that have an associated run by counting the number of 1-bits in θ up to q. Equation 3.4 then finds the position of q's runend by using SELECT to find the position of the m-th 1-bit in ρ.

Given the end of the run, s, it is easy to walk backwards through the rest of the run. We reach the start of the run when we have reached either the end of another run or the q-th slot in the RSQF. An RSQF invariant is that a remainder is never placed before its quotient; the latter condition relies on this invariant. In other words, the run begins at a position s_0 if either of the two conditions hold:

1. $s_0 \neq s$ and $\rho[s_0 - 1] = 1$; or

2. $s_0 = q$.

If the remainder rem(x) is discovered during the traversal, i.e., $\exists\ i$ where $s_0 \leq i \leq s$ and $\sigma[i] = \text{rem}(x)$, then the RSQF returns PRESENT. Otherwise, it returns ABSENT.

The query algorithm is presented in Algorithm 3.1. The algorithm first checks the occupancy of $\theta[\text{quot}(x)]$, returning ABSENT if the bit is unset. If it is set, the algorithm uses RANK and SELECT to find the end of quot(x)'s run. It then steps through the run, terminating when the run's termination conditions are met ($i < \text{quot}(x)$ or $\rho[i] = 1$). If a matching remainder is found in the run, the algorithm returns PRESENT; otherwise, it returns ABSENT.

```
function LOOKUP(Q, x)
    if θ[quot(x)] = 0 then
        return ABSENT
    m ← RANK (θ, quot(x))
    i ← SELECT (ρ, m)
    repeat
        if σ[i] = rem(x) then
            return PRESENT
        i ← i − 1
    until i < quot(x) or ρ[i] = 1
    return ABSENT
```

Algorithm 3.1: Query algorithm for simple RSQF. [16]

3.2.2 Insertions

At a high level, inserting a new fingerprint $f(x) = q \circ r$ into an RSQF $Q = (\theta, \rho, \sigma)$ is a matter of placing the remainder r at the end of q's run, or starting a new run for q if none exists. This can be broken down into the following steps:

```
1 if q does not have an associated run then
2     Make room for r at q by shifting over prior occupants.
3     Insert r into σ[q], adjusting metadata as necessary.
4 else
5     Find the end of q's run, s.
6     if σ[s] is occupied by another remainder then
7         Make room for r at s by shifting prior occupants forward.
8     Insert r into σ[s], adjusting metadata as necessary.
```

Determining whether q has an associated run (Line 1) is straightforward—we simply check the occupied bit for q, $\theta[q]$. In either case, the RSQF must ensure that all quotient-sharing remainders are stored in a single contiguous run. To maintain this invariant, it is sometimes necessary to shift over some remainders to make room for the new remainder r (Lines 2 and 7).

If remainders need to be shifted to make room, then the range of remainders to shift over is $[x, u-1]$, where x is the desired insertion site for the new remainder and u is the first empty slot after x. Shifting each remainder in $[x, u-1]$ forward by one slot preserves the RSQF's invariants and frees up slot x.

Any runend bits in the range $[x, u-1]$ must also be shifted over. Occupied bits do not need to be modified, as shifting a quotient's run does not affect whether the quotient has a run.

To find u, we define the function FirstUnused(Q, i), which finds the index of the first unused remainder slot in σ after position $i-1$. (If slot $\sigma[i]$ is unoccupied, FirstUnused(Q, i) returns i.) The FirstUnused function can be defined using rank and select (Figure 3.2).

```
function FirstUnused(Q, i)
    m ← Rank(θ, i)
    s ← Select(ρ, m)
    while i ≤ s do
        i ← s + 1
        m ← Rank(θ, i)
        s ← Select(ρ, m)
    return i
```

Algorithm 3.2: Algorithm to find the first unused slot after i in an RSQF Q. [16]

Placing a remainder into an empty slot is simple. If the new remainder is the first in its run, then its quotient q must be marked as occupied ($\theta[q] \leftarrow 1$) and the q-th runend bit must be set to 1 ($\rho[q] \leftarrow 1$). If the new remainder r being inserted at slot i is not the first in its run, then the $(i-1)$-th runend bit must be shifted forward ($\rho[i-1] \leftarrow 0$ and $\rho[i] \leftarrow 1$). The quotient q's occupied bit is already set to 1, and does not need to be modified.

Before proceeding to the full algorithm, a final remark: let us consider the rank and select

operations used in the query algorithm in more detail (Equations 3.3 and 3.4, reproduced below).

$$m = \textsc{rank}(\theta, q)$$
$$s = \textsc{select}(\rho, m).$$

In the query algorithm, r and s were computed only when the quotient q was occupied, so s stored the end of q's run. We did not previously consider the case where q was unoccupied.

When q is unoccupied, $\textsc{rank}(\theta, q)$ counts the number of occupied bits in $[0, q-1]$. In other words, m tracks the number of occupied quotients before q. Then, $\textsc{select}(\rho, m)$ locates the runend bit corresponding to m, the number of occupied quotients before q. Therefore, when q is unoccupied, s stores the runend of the closest occupied quotient before q.

Knowing s is useful. RSQF runs are contiguous and ordered by quotient: the run for a quotient q_0 comes before the run for some q_1 where $q_0 < q_1$. When the fingerprint $q_1 \circ r_1$ is being added to the RSQF, it is important that r_1 is inserted after all of the remainders in q_1's run. If q is *unoccupied*, then, we must insert r after s to avoid overwriting the runs of any smaller quotients. When q is *occupied* instead, we still insert r after s, although for a different reason—s stores q's runend.

We present the full insertion algorithm in Algorithm 3.3.

```
function Insert(Q, x)
    m ← Rank(θ, quot(x))
    s ← Select(ρ, m)
    if quot(x) > s then
        σ[quot(x)] ← rem(x)
        ρ[quot(x)] ← 1
    else
        s ← s + 1
        u ← FirstUnused(Q, s)
        while u > s do
            σ[u] ← σ[u − 1]
            ρ[u] ← ρ[u − 1]
            u ← u − 1
        σ[s] ← rem(x)
        if θ[quot(x)] = 1 then
            ρ[s − 1] ← 0
        ρ[s] ← 1
    θ[quot(x)] ← 1
```

Algorithm 3.3: Algorithm to insert an element into the simplified RSQF. [16]

3.2.3 Summary

We have defined a simplified version of the RSQF which consists of a single remainder array and two metadata bitarrays storing *occupied* and *runend* bits. In this simplified RSQF, insertions and queries can be answered correctly using the information in these metadata bitarrays, with the help of the RANK and SELECT operations that scan each metadata bitarray in full.

This simplified RSQF is slow. Performing a single insertion or query will require traversing the

entire remainder array. Furthermore, our RANK and SELECT implementations operate on 64-bit words, not n-bit bitarrays.

In the next section, we discuss the blocked RSQF, which segments the remainder array and metadata bitarrays into 64-element chunks. This improves locality and allows the use of our fast RANK and SELECT operations, developed in Section 3.1.2.

3.3 Blocked RSQF

To improve cache efficiency, we can split the RSQF's remainder array and its two metadata bitarrays into 64-element blocks. We can then consider the RSQF as an array of blocks, where each block stores a 64-element remainder array and two 64-bit metadata bitarrays. This allows us to use fast RANK and SELECT operations on 64-bit words to quickly operate on the metadata of each block.

Figure 3.5 shows how the blocked RSQF differs from the unblocked RSQF in structure. The blocked RSQF is stored as an array of pointers, each of which reference a 64-element block.

3.3.1 Offsets

Introducing offsets

To obviate the need to search through the entire filter when looking for a run, Pandey et al. [16] add an *offset* field to each block. The *logical offset* τ of a location i is defined as the distance from i to the runend of the closest occupied quotient $q \leq i$. The *actual offset* $\hat{\tau}$ (shortened to "offset"), which is stored in the offset field of each block, is a non-negative version of the logical offset.

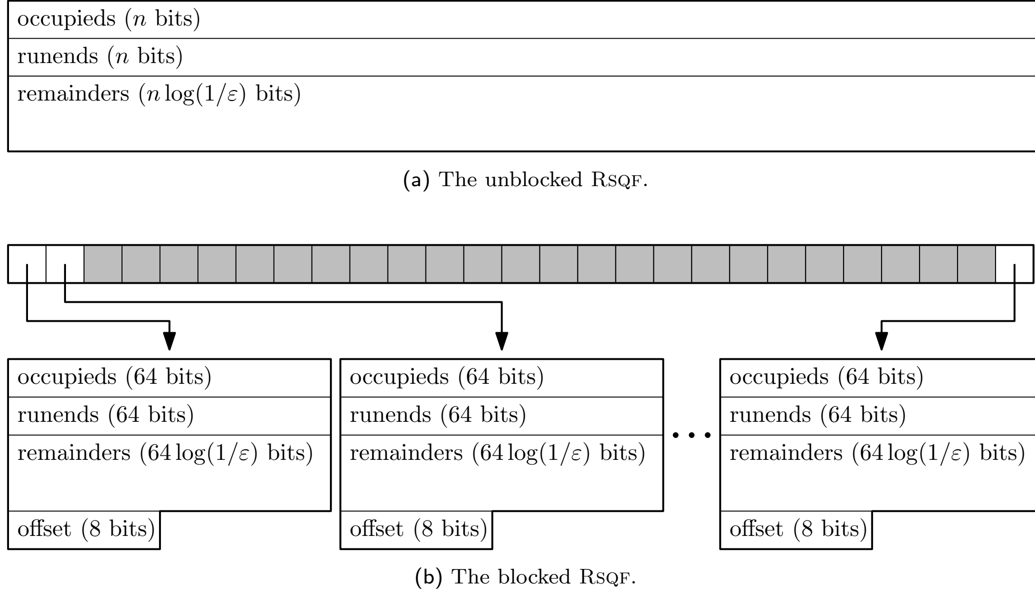
$$\tau(i) = \text{SELECT}(\rho, \text{RANK}(\theta, i)) - i \tag{3.5}$$

$$\hat{\tau}(i) = \max\{0, \tau(i)\} \tag{3.6}$$

When the quotient i is occupied ($\theta[i] = 1$), the closest occupied quotient is i itself, so $\hat{\tau}(i)$ stores the distance between i and i's runend. When i is unoccupied ($\theta[i] = 0$), the closest occupied quotient is some $j < i$, so $\hat{\tau}(i)$ refers to the distance between i and j's runend. If j's runend comes before i, then the offset will be negative. In this case, we store an offset of 0 instead of storing the negative distance (hence the $\max\{0, \cdot\}$).

The RSQF stores an offset for the first slot in each block. In other words, the i-th block in an RSQF stores $\hat{\tau}(64 \cdot i)$ in its offset field. Including an offset field in each block allows us to determine which of the runend bits stored in a block belong to quotients (occupied bits) in the block. This is important: without the offset field, it would be impossible to determine which quotient a runend belongs to without traversing all previous blocks.

The inclusion of offset fields makes it possible to use RANK and SELECT operations to find runs without scanning through the entire array. Instead, insertions and queries can start at the block containing the target quotient and scan through a constant number of blocks, with high probability. This improves runtime significantly, but also substantially increases the complexity of queries and insertions. In particular, offsets must be carefully maintained. Anytime runends are shifted around (i.e., during insertion time), the offsets pointing to them must also be adjusted.



(a) The unblocked Rsqf.

(b) The blocked Rsqf.

Figure 3.5: Comparing the simple (unblocked) Rsqf to a blocked Rsqf.

Offsets can be restricted to 8 bits without affecting performance [16]. This brings the Rsqf's total metadata space usage to 2.125 bits per element (2 bits per slot, 8 bits per block).

Negative logical offsets

While actual offsets do not distinguish between zero offsets and negative offsets, it is useful to distinguish negative logical offsets from non-negative logical offsets. The sooner we can determine that a logical offset is negative, the sooner we can discard it from consideration (and return EMPTY). By taking into account other block metadata, we can determine whether a block's logical offset is negative.

The following conditions must hold for the i-th block $(\theta, \rho, \sigma, \hat{\tau})$ to have a negative logical offset:

$$\theta[0] = 0 \text{ and } \hat{\tau} = 0 \text{ and } \rho[0] = 0 \tag{3.7}$$

1. $\theta[0] = 0$. The first slot in the block must be unoccupied. If it were occupied, then $\hat{\tau}$ would point to the runend of the quotient $64i$.

2. $\hat{\tau} = 0$. This is true by definition (Equation 3.6). If $\hat{\tau} > 0$, then $\max\{0, \tau\} = \hat{\tau} > 0$, so $\tau > 0$.

3. $\rho[0] = 0$. The first runend in the block must be unset. If it were set, with $\theta[0] = 0$ and $\hat{\tau} = 0$, then τ would be point to $\rho[0]$ and would be 0, not negative.

Updating offsets

Recall that at insertion time, an interval $[a, b]$ of remainders is shifted forward to make room for new remainders, and runends are shifted with the remainders. Offsets point to runends; therefore, insertions must also update any offsets that point to runends in $[a, b]$.

To handle these offset updates, we introduce UpdateOffsets (Algorithm 3.4) and UpdateOffsetsNewRun (Algorithm 3.5). UpdateOffsets increments offsets pointing to runends within a specified range; UpdateOffsetsNewRun also increments offsets, but has slightly different behavior to accommodate a new run. Note that both functions account for negative offsets (Lines 7 and 6).

From this point onward, unless otherwise specified, we use the division operator as a shorthand for *truncating division*: the expression a/b actually means $\lfloor a/b \rfloor$.

```
1  function UpdateOffsets(Q, a, b)
2      if a > b then return
3      i ← min{b/64 + 1, |Q| − 1}
4      while i ≥ 0 do
5          (θ, ρ, σ, τ̂) ← Block(Q, i)
6          i_b ← 64 · i
7          if θ[0] = 0 and τ̂ = 0 and ρ[0] = 0 then
8              continue
9          t ← i_b + τ̂
10         if t < a then return
11         else if t ≤ b then
12             τ̂ ← τ̂ + 1
13         i ← i − 1
```

Algorithm 3.4: A helper function that shifts offsets pointing to runends in $[a, b]$.

```
1  function UpdateOffsetsNewRun(Q, q, i)
2      j ← min{i/64 + 1, |Q| − 1}
3      while j ≥ 0 do
4          (θ, ρ, σ, τ̂) ← Block(Q, j)
5          i_b ← 64 · j
6          if θ[0] = 0 and τ̂ = 0 and ρ[0] = 0 then
7              continue
8          t ← i_b + τ̂
9          if t < i then return
10         else if t = i and θ[0] = 0 and q ≤ i_b then
11             τ̂ ← τ̂ + 1
12         j ← j − 1
```

Algorithm 3.5: A helper function that shifts offsets pointing to i, the site of a remainder insertion for the quotient q.

3.3.2 Shifting metadata across blocks

Shifting remainders and runends was trivial in the simple Rsqf. It is fundamentally the same for the blocked Rsqf, with some minor differences to account for block boundaries.

3.3.3 Abstractions for rank and select

Rsqf membership queries must take block offsets into account when using rank and select to find a quotient's runend. To abstract away reasoning about offsets and blocking in queries and insertions,

```
function ShiftRemsAndMetadata(Q, a, b)
    if a > b then return
    i ← b
    while i ≥ a do
        (_, ρ, σ, _) ← Block(Q, i/64)
        ρ[(i + 1)/64] ← ρ[i/64]
        σ[(i + 1)/64] ← σ[i/64]
        i ← i − 1
    (_, ρ, _, _) ← Block(Q, a/64)
    ρ[a] ← 0
```

Algorithm 3.6: Shifting remainders and runends for the blocked Rsqf.

we introduce the following helper functions:

1. SelectRunend(Q, i_b, r) finds the index of the r-th runend after the start of the i_b-th block.

2. RankSelect(Q, x) performs the blocked equivalent of the rank and select operations in Equations 3.3 and 3.4. That is, the result of RankSelect is the equivalent of select$(\rho, \text{rank}(\theta, x))$ in a blocked context. In other words, RankSelect finds the runend of q, the closest quotient to x (q may equal x):

$$q = \max\{0 \leq i \leq x \mid \theta[i] = 1\}$$

   When RankSelect(Q, x) gives a negative result, the precise value does not matter for either insertions or queries. A negative result means that the slot $\sigma[x]$ is unoccupied; this information alone is sufficient. To indicate that a slot is unoccupied, RankSelect returns empty. This allows RankSelect to save time and exit early when it detects that the slot is unoccupied.

```
function SelectRunend(Q, i_b, r)
    i ← i_b × 64
    loop
        (θ, ρ, σ, τ̂) ← Block(Q, i/64)
        s ← select (ρ, r)
        i ← i + s
        if s ≠ 64 then return i        ▷ select returns 64 when there are more
                                         than r set bits in ρ
        else if i ≥ |Q| then return −1
        else
            r ← r − popcnt(ρ)
```

Algorithm 3.7: Blocked runend-finding algorithm for Rsqf.

3.3.4 Queries

With RankSelect in hand, we can adapt the simple Rsqf's query algorithm for the blocked Rsqf. The query algorithm is presented in Algorithm 3.9; it is almost unchanged from Algorithm 3.1.

```
function RankSelect(Q, x)
    i_b ← x/64
    i_s ← x mod 64
    (θ, ρ, σ, τ̂) ← Block(Q, i_b)
    if θ[0] = 1 and τ̂ = 0 and ρ[0] = 0 then
        if i_s = 0 then return EMPTY
    else
        if i_s = 0 then return 64 · i_b + τ̂
        else
            i_b ← i_b + τ̂/64
    d ← RANK(θ, i_s) − θ[0]
    t ← τ̂ mod 64
    (θ, ρ, σ, _) ← Block(Q, i_b)
    d ← d + RANK(ρ, t)
    if d = 0 then return EMPTY
    else
        j ← SelectRunend(Q, i_b, d − 1)
        if j < x then return EMPTY
        else return j
```

Algorithm 3.8: Performs the blocked equivalent of Equation 3.4, with some optimizations to exit early.

3.3.5 Insertions

Insertions require a little more work. In particular, blocked RSQF insertions require incrementing offsets in response to any shifted runends (recall that runends are shifted along with remainders). We make use of our offset-maintenance functions introduced earlier in the insertion algorithm (Algorithm 3.10).

3.4 Summary

The RSQF is a performant quotient filter that uses fast RANK and SELECT operations on 64-bit words to quickly find the run associated with a given quotient. The RSQF stores two metadata bitarrays, *occupieds* and *runends*, to determine how remainders fit into runs.

We have presented two views of the RSQF. The first simplifies the RSQF into a single large array to demonstrate how RANK and SELECT may be used to find runends. The second splits the RSQF into 64-element blocks to improve cache efficiency and enable the use of our fast RANK and SELECT operations. Blocking introduces additional complexity, primarily from the inclusion of an *offset* field that enables searches to examine a constant number of blocks with high probability instead of scanning through the RSQF from the beginning.

The RSQF uses 2.125 metadata bits per element: 2 bits per slot for each bitarray (θ, ρ) and 8 bits per (64-element) block for each block's offset $(\hat{\tau})$.

```
function Lookup(Q, x)
    if θ[quot(x)] = 0 then
        return absent
    i ← RankSelect(Q, x)
    if i = empty then return absent
    repeat
        if σ[i] = rem(x) then
            return present
        i ← i − 1
    until i < quot(x) or ρ[i] = 1
    return absent
```

Algorithm 3.9: The blocked Rsqf's query algorithm.

```
function Insert(Q, x)
    i ← RankSelect(Q, quot(x))
    if i = empty then
        θ[quot(x)] ← 1
        ρ[quot(x)] ← 1
        σ[quot(x)] ← rem(x)
    else
        u ← FirstUnused(Q, s)
        UpdateOffsets(Q, i + 1, u − 1)
        ShiftRemsAndMetadata(Q, i + 1, u − 1)
        if θ[quot(x)] = 1 then
            UpdateOffsets(Q, i)
            ρ[i] ← 0
            ρ[i + 1] ← 1
            σ[i + 1] ← rem(x)
        else
            UpdateOffsetsNewRun(Q, quot(x), i)
            θ[quot(x)] ← 1
            ρ[i + 1] ← 1
            σ[i + 1] ← rem(x)
        ρ[s] ← 1
    θ[h_0(x)] ← 1
```

Algorithm 3.10: The blocked Rsqf's insertion algorithm.

Chapter 4

Arithmetic Coding

Constructing an adaptive filter entails storing extra information to fix past false positives. But storing extra information in a filter is costly. Making the filter bigger to accommodate extra data will lead to slower access times; keeping the filter the same size and budgeting bits toward tracking prior false positives will leave fewer bits for the base filter, leading to more frequent false positives on new inputs. Therefore, it is critical that an adaptive filter should use minimal space to remember past false positives.

Luckily, storing information succinctly is a well-studied problem. After all, it is the goal of compression. Compression algorithms succinctly represent messages by using fewer bits to represent frequently occurring values and more bits to represent infrequently occurring values. This minimizes the length of the average compressed message, because high-frequency values take little space once compressed, and values with large encodings occur rarely.

To store adaptivity information effectively, we need a compression scheme that satisfies the following requirements:

1. the code should use close to optimal space;
2. the code should be able to use, on average, < 1 bit per encoded symbol;
3. the code should optimize for a highly skewed distribution, where a single symbol has probability close to 1; and
4. encoding and decoding operations should be fast enough to avoid significantly slowing down the filter.

Huffman coding is fast and close to entropy optimal, but cannot encode characters with < 1 bit per space. Moreover, it performs poorly on highly skewed probability distributions [21]. *Arithmetic coding*, on the other hand, is optimal in theory and very close to optimal in practice, even with fractional bits [9]. Arithmetic coding also outperforms Huffman coding for skewed probabilities [9, 21]. Its primary drawback is that it is very slow.

In this work, we introduce a modified arithmetic coding scheme that achieves fast practical performance by tailoring the scheme's encoding and decoding operations to the task at hand.

In this chapter, we introduce arithmetic coding and discuss general heuristics for improving its throughput performance. Further discussion of tailoring arithmetic coding to specific filter implementations is deferred to Chapter 5.

4.1 Introducing arithmetic coding

In an arithmetic code, a message, consisting of a string of symbols, is represented as an interval. A message's interval is long or short in proportion to how probable the message's contents are: a highly probable message will be coded as a long interval, and an improbable message will be coded as a short interval. Each interval is in turn represented by a number in the interval. Picking a number from a longer interval requires less precision than picking one from a shorter interval, so longer intervals can be represented with fewer bits than shorter intervals. For example, given the interval $[0.1, 0.2]$, it is sufficient to store 0.1, whereas for the narrower interval $[0.11, 0.12]$, we must store 0.11, which takes more space.

In summary: Messages with high-probability symbols have longer intervals, which require fewer bits to represent. Therefore, high-probability messages require fewer bits to represent than low-probbaility messages. This is how arithmetic coding achieves good compression.

4.1.1 Encoding

More precisely, arithmetic coding represents a message comprised of the letters $l_1, l_2, ..., l_n$, each taken from the alphabet $\Sigma = \{a_1, a_2, ..., a_m\}$, as a subinterval of the unit interval, $[0, 1]$. At a high level, the procedure works as follows [9]:

1. Start with a "current interval" $[L, H)$ initialized to $[0, 1)$.
2. Partition the current interval into subintervals, one for each letter in Σ, where the length of the i-th subinterval is proportional to its probability, $\Pr(a_i)$.
3. Select the subinterval corresponding to the next letter in the message, and treat it as the new current interval.
4. Continue to partition and select subintervals until the last letter of the message has been reached. Its corresponding subinterval, $[s, t]$, is our result; we represent it by s, t, or some other number in between.

Figure 4.1 illustrates the encoding process for an example alphabet $\Sigma = \{a, b, c\}$ and message "*abc*", where $\Pr(a) = \frac{1}{2}$ and $\Pr(b) = \Pr(c) = \frac{1}{4}$. Figure 4.1(a) illustrates the process without modification; Figure 4.1(b) scales up each subinterval to make each subdivision more readily apparent.

With the idea behind arithmetic coding in hand, we now present a naïve encoding algorithm (Algorithm 4.1). The algorithm maintains the low and high ends (L and H, respectively) of the current interval, and incrementally narrows the interval by adjusting the values of L and H.

function ENCODE(m)
  $L \leftarrow 0$
  $H \leftarrow 1$
  for each $a_i \in m$ **do**
    $r \leftarrow H - L$
    $L \leftarrow L + r \cdot \left(\sum_{j=1}^{i-1} \Pr(a_j)\right)$
    $H \leftarrow L + r \cdot \Pr(a_i)$
  return L

Algorithm 4.1: Encoding a message m.

When encoding, we associate with each letter $a_i \in \Sigma$ the following interval:

$$\mathrm{I}(a_i) = \left[L + (H - L) \cdot \sum_{j=1}^{i-1} \Pr(a_j), L + (H - L) \cdot \sum_{j=1}^{i} \Pr(a_j) \right]$$

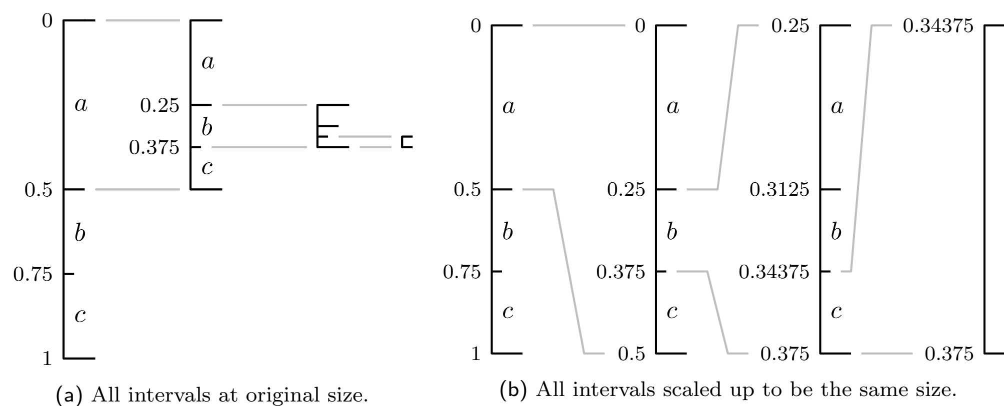
Note that the length of the interval is $|I(a_i)| = (H - L) \cdot \Pr(a_i)$. Therefore, subdividing the interval $[L, H)$ and taking the piece associated with the letter a_i effectively multiplies the length of $[L, H)$ by $\Pr(a_i)$. So the length of the interval associated with a message $m = l_1, ..., l_n$ equals the product of the probabilities of each letter in the message:

$$|\mathrm{ENCODE}(m)| = \prod_{a_i \in m} \Pr(a_i)$$

This makes it clear that arithmetic coding naturally leads to a very nearly optimal code in practice: $-\log p$ bits are used to encode a symbol whose probability of occurrence is p.

4.1.2 Decoding

Decoding an arithmetic code is simple. Recall that a message is encoded as a point p in $[0, 1)$. For simplicity, let us assume that we know the length l of the encoded message.



(a) All intervals at original size.

(b) All intervals scaled up to be the same size.

Figure 4.1: An example of the encoding process. Reproduced in part from [21].

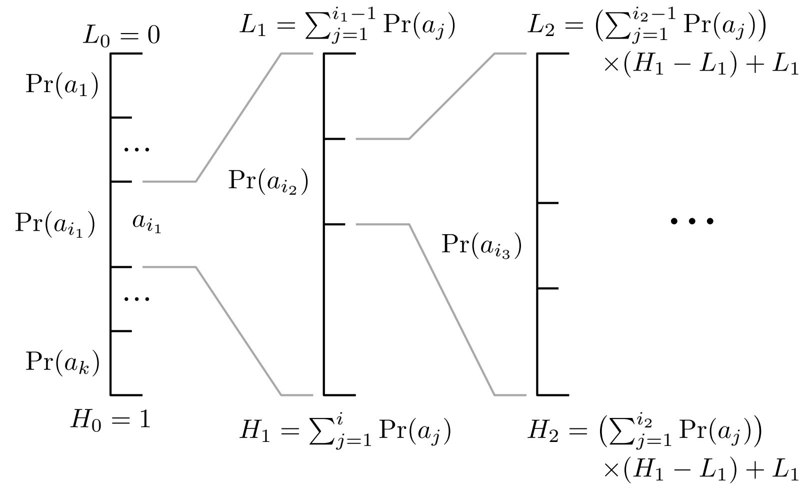


Figure 4.2: Encoding a message m, visually.

1. Start with a current interval $[L, H)$ initialized to $[0, 1)$.
2. Partition the current interval into subintervals, one for each letter in Σ, where each subinterval's length is proportional to the probability of the letter it is associated with.
3. Select the subinterval containing p as our new current interval.
4. Continue to partition and select subintervals until l letters have been decoded.

Figure 4.3 illustrates this process for the alphabet and probability distribution used in our previous example. Figure 4.4 presents pseudocode.

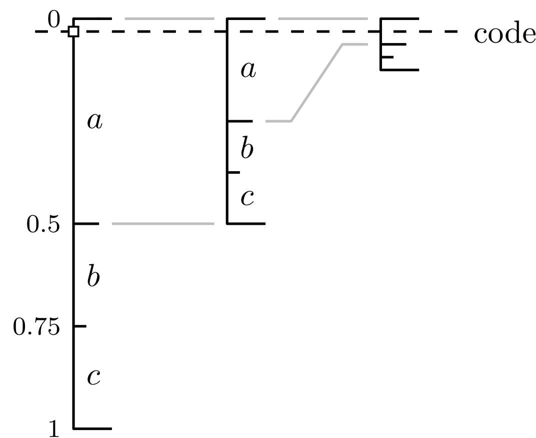


Figure 4.3: Decoding the point 0.1 as the message "aaa".

Note that the encoding from the prior example (Figure 4.3) can be decoded as several different messages if we do not know the message's length in advance. If the message is three letters long, the message must be "aaa". But if it is four letters long, the message will be "aaac". While we

```
function DECODE(p, l)
    L ← 0
    H ← 1
    for i ∈ 1, ..., l do
        k ← FINDSYMBOL(p, L, H)
        r ← H − L
        H ← L + r · (∑_{i=1}^{k} Pr(a_i))
        L ← L + r · (∑_{i=1}^{k−1} Pr(a_i))
    return L
```

```
function FINDSYMBOL(p, L, H)
    s ← L
    r ← H − L
    for i ∈ 1, ..., |Σ| do
        if s + r · Pr(a_i) ≥ p then
            return i
        else
            s ← s + r · Pr(a_i)
    return n
```

Figure 4.4: Decoding an arithmetic code $p \in [0, 1)$ for a message of length l. We use the subroutine FindSymbol to select the subinterval in the current interval that contains p.

assumed previously that we have foreknowledge of the message's length, often the only artifact of an arithmetic code is the code itself, a single floating-point number. To store the message's length in the encoding, we add an extra letter `$` to our alphabet to denote the end of the message [21]. Using this approach, we would encode "aaa" as "aaa`$`" and "aaac" as "aaac`$`". This increases the space cost of our encoding, as we must reserve room for `$` in the probability distribution.

As presented thus far, arithmetic coding requires the use of floating-point operations, as subdividing intervals uses floating-point multiplications and divisions. These operations are slow. They also introduce precision issues [13]. In the next section, we discuss how we can avoid these issues by doing away with floating points entirely. For one, instead of using the unit interval, we can use an interval of integers, $[0,\ 2^k - 1]$, where k is the desired encoding length. In addition, we use a mix of bit shifts and additions to quickly approximate multiplications and divisions, trading a bit of accuracy for speed.

4.2 Fast approximate arithmetic codes

Arithmetic coding gives optimal compression in theory and very close to optimal compression in practice [9]. But on the unit interval, it relies on floating point operations, which are slow and can introduce imprecision. For a naïve arithmetic coding implementation, these floating point operations are needed to subdivide intervals (which, being subintervals of the unit interval, need floating point numbers to represent) using probabilities (which are also between 0 and 1).

We avoid these issues by introducing two changes. First, we replace the unit interval with an interval of integers, $[0,\ 2^k - 1]$ for the desired code length k. We only use integers to represent further subdivisions of this interval. Second, when multiplying or dividing integral intervals by probabilities, we approximate these floating point operations using bit shifts and additions. In effect, we approximate each multiplication operation using a sum of inverse powers of 2.

For example, to approximate the value 0.782, we can use the sum $1/2 + 1/4 + 1/32 = 0.78125 \approx 0.782$. Multiplications by 0.782 can then be approximated using bit shifts (Algorithm 4.2).

Importantly, approximating a probability p with a value $\hat{p}$ less than p does not introduce correctness issues, but using $\hat{p} > p$ does. When $\hat{p} > p$, the subinterval associated with p overlaps with

```
function Mult0.782(x)
    p ← (x >> 1) + (x >> 2) + (x >> 5)
    return p
```

Algorithm 4.2: Approximately multiplying x by 0.782 using bit shifts and additions.

the subinterval of another letter, introducing ambiguity that breaks the code. In the former case, when $\hat{p} < p$, the code loses some space to imprecision, but this does not affect correctness because using a smaller subinterval does not introduce ambiguity.

Further discussion of this method for specific filter implementations is presented in Section 5.2.2.

As a final optimization for our particular use case, we do not encode the "message-end" symbol `$`, as all blocks have a fixed number of elements (64). We hard-wire our decode operation to stop after reading 64 symbols. This allows us to store more information in our codes as we do not need to reserve space for `$` in the probability distribution for symbols.

4.3 Summary

Arithmetic coding is a theoretically optimal and practically close-to-optimal compression technique that represents a string of symbols as an interval. Each letter in the alphabet, along with its associated probability, are associated with a piece of each interval. Encoding recursively subdivides a starting interval, representing a letter with each subdivision. The result is a single point in the interval. Decoding recovers the message's contents by identifying the subintervals that this point belongs to.

The general, naïve implementation uses floating point multiplication to subdivide the unit interval with symbol probabilities. Floating point operations slow down arithmetic coding operations and introduce imprecision. We introduce two changes to remedy these problems. First, we replace the unit interval with an integral interval. Second, we replace floating point multiplications using bit shifts and additions. Our modifications make arithmetic coding fast enough to be used in practice.

Chapter 5

Practical Adaptive Quotient Filters

In this chapter, we first introduce the broom filter [2], the only known theoretically adaptive filter. We then discuss two practical mechanisms for adaptivity: fingerprint extensions and hash selectors. To make these mechanisms practical, we use fast arithmetic coding to store extensions and selectors quickly and succinctly. We also discuss a simple remote representation for quotient filters that enables easy reverse lookups of elements from their fingerprints.

Next, we describe our primary contribution: the extension-based adaptive quotient filter (ExtQF) and the selector-based adaptive quotient filter (SelQF). The ExtQF is a practical implementation of the broom filter that uses arithmetic coding to store fingerprint extensions succinctly. The SelQF is a close relative of the ExtQF that resolves false positives by changing fingerprints instead of extending them. When a hash collision (false positive) occurs, the SelQF updates the fingerprint that caused the collision by rehashing it with a new hash function, thereby lowering the probability that the false positive recurs. To identify which hash function was used, the SelQF stores an identifier for the function. We call this identifier a *hash selector*; this gives the SelQF its name.

5.1 Broom filter

First, we reintroduce the broom filter in more depth. The broom filter is a theoretical filter; it does not yet have an implementation suitable for practical use [2, 4]. Nonetheless, it has strong theoretical adaptivity guarantees [2].

5.1.1 Definition and remote representation

The broom filter consists of two parts: a *local filter* L and a *remote oracle* R. The local component L is a conventional non-adaptive filter with $O(n)$ extra space for adaptivity; for our purposes, we can think of L as an augmented quotient filter, although it could just as well be any other kind of single hash function filter [15].

The remote component R provides extra information used to modify L, making the broom filter $B = (L, R)$ adaptive. In particular, when a false positive occurs on fingerprint in L, R tells L which element in the set represented by L corresponds to the faulty fingerprint. This allows B to adjust the internal state of L to ensure that the false positive is not repeated. Below, we present the core

operations associated with a broom filter (Definition 5.1).

Definition 5.1 (Broom filter)**.** A broom filter $B = (L, R)$, representing the set S with false positive rate ε, has the following deterministic functions:

- $\textsc{Init}(n, \varepsilon) \to (L, R)$. Creates a new broom filter of size n.
- $\textsc{Lookup}(L, x) \to (L', y)$. Checks whether $x \in U$ is in L, reporting the result in $y \in \{\textsc{present}, \textsc{absent}\}$. Modifies L in the case of a false positive to yield L'.
- $\textsc{Insert}(L, R, x) \to (L', R')$. Inserts an element $x \in U$ into B.
- $\textsc{Adapt}(L, R, x) \to (L', R')$. Fixes a false positive x.

Lookup does not have access to R: this ensures that queries only need to access the small local representation, reducing query times. Insert modifies both L and R. Adapt fixes a false positive by modifying L and R. In other words, for some $x \notin S$ where $\textsc{Lookup}(L, x)$ returns present, $\textsc{Lookup}(L', x)$ will return absent.

The broom filter's approach of partitioning its data into two states L and R helps manage the balancing act between robust adaptivity properties and practical performance. L is a filter, so it is small and can quickly respond to queries. R contains extra information needed to ensure adaptivity, so that $B = (L, R)$ can meet theoretical adaptivity guarantees.

In fact, this two-part architecture extends to all adaptive filters. Bender et al. [2] show that any adaptive filter must use at least $\Omega(\min\{n \log\log u,\ n \log n\})$ bits to represent a set of n elements drawn from a universe of u elements. Storing this information in one place would defeat the purpose of using a filter, because $\Omega(\min\{n \log\log u,\ n \log n\})$ is very close to $\Omega(n \log u)$ bits, at which point it makes more sense to store S directly. Partitioning the filter's state into two components allows adaptive filters to preserve the performance characteristics of non-adaptive filters in the local state, while maintaining a remote state that contains information needed to guarantee robust adaptivity properties.

5.1.2 Fingerprint extensions

The broom filter is a single-hash-function filter [15], i.e., it stores fingerprints for each element in S. Each fingerprint $f(x)$ is a prefix of a longer hash $h(x)$, denoted $f(x) \sqsubseteq h(x)$. A false positive occurs on a query $y \notin S$ when there exists a stored element $x \in S$ such that $f(x) = f(y)$ and $y \neq x,\ x \in S$. The broom filter resolves this collision by adding bits to $f(x)$ until it no longer collides with $f(y)$. In other words, the filter updates $f(x)$ to be the shortest prefix of $h(x)$ that is not a prefix of $h(y)$. Let $\mathrm{pre}(s, k)$ denote the k-letter prefix of the string s. Then the new fingerprint $g(x)$ is the following:

$$
\begin{aligned}
g(x) &= \mathrm{pre}(h(x), k) \\
k &= \min\big\{l \in |f(x)|, ..., |h(x)| \mid \mathrm{pre}(h(x), l) \neq \mathrm{pre}(h(y), l)\big\}
\end{aligned}
$$

Extending $f(x)$ by k bits not only fixes the collision with y, but also reduces the probability of all future collisions on $f(x)$ by 2^{-k}. This gives rise to a phenomenon that Bender et al. term

"serendipitous corrections" [4]: a correction to a single collision reduces the probability of all future collisions with $f(x)$.

Fingerprints cannot be allowed to grow without bound, however, as unrestricted fingerprint extensions would inflate the filter's size. To keep the filter small and performant, extension length must be bounded, and extension bits may need to be reclaimed over time as long fingerprints grow stale. The broom filter uses $O(n)$ extension bits and rebuilds with a new hash after $O(n)$ queries. This has the potential to reduce the impact of serendipitous corrections, as serendipity has limited time to manifest itself. We discuss this issue in Sections 5.1.4 and 5.3.5.

Like a quotient filter, the broom filter splits its fingerprints into multiple parts. The first $\log n$ bits constitute the quotient; the next $\log(1/\varepsilon)$ bits are the remainder; and the remaining bits, whose length varies depending on the fingerprint's collision history, are called "adaptivity bits" by the authors of the broom filter. For our purposes, we will call these extra bits *extension bits* to disambiguate them from other mechanisms for adaptivity.

In the discussion that follows, we will use $\text{quot}(x)$, $\text{rem}(x)$, and $\text{ext}(x)$ to denote respectively the quotient, remainder, and extension bits used to represent the element x, and $f(x)$ to denote the whole fingerprint stored for x, where $f(x) = \text{quot}(x) \circ \text{rem}(x) \circ \text{ext}(x)$, and $a \circ b$ denotes the concatenation of a and b.

Notation	Meaning	Length
$\text{quot}(x)$	x's quotient	$\log n$
$\text{rem}(x)$	x's remainder	$\log 1/\varepsilon$
$\text{ext}(x)$	x's extension	$O(1)$
$f(x)$	x's full fingerprint	$\log n/\varepsilon + O(1)$

Figure 5.1: Notation for fingerprint pieces.

5.1.3 Core operations

Insertions and queries are essentially the same as they are for a quotient filter, with some modifications to maintain extension bits for each fingerprint in the filter and accommodate R, the remote representation. Adaptive filters have an additional core operation, ADAPT, which fixes a false positive. Because the broom filter is a theoretical filter defined in terms of L and R, we consider all operations at a high level.

Insertions

Inserting an element x into a broom filter $B = (L, R)$ modifies both L and R. Recall that insertions to a quotient filter use the quotient as a kind of index that marks roughly where the remainder is stored; taken together, the quotient and remainder reconstruct the element's full fingerprint. The only difference between inserting x into a conventional quotient filter and inserting it into a broom filter's local state L is that L associates extension bits with each remainder. Effectively, L treats $\text{quot}(x)$ as the quotient and $\text{rem}(x) \circ \text{ext}(x)$ as the remainder.

Inserting x into B also requires inserting x into R. Because this operation depends on the implementation of R, we defer this discussion to Section 5.3.1.

Queries

To check whether x is in B, we simply check the membership of x in L via L's query operation, taking extension bits into account. Equivalently, B reports PRESENT for x if there exists a fingerprint $f(y)$ in L such that $f(y) \sqsubseteq h(x)$. Note that we do not check R—if we did, all queries would go straight to the remote representation. This would defeat the purpose of using a filter.

Adapts

Consider a false positive instance: let $x \in S$ and $y \notin S$ such that $f(x) \sqsubseteq h(y)$. To fix this false positive, the broom filter extends the fingerprint $f(x)$, yielding a new fingerprint $g(x)$ such that $g(x) \not\sqsubseteq h(y)$. To perform this extension, however, the broom filter needs to be able to recover x from $f(x)$. This is the role of R.

5.1.4 Bit reclamation

Adapting will gradually add bits, increasing the size of the broom filter. Bender et al. suggest reclaiming bits by changing the filter's hash function upon every $\Theta(n)$-th call to ADAPT and present a way of deamortizing this process. The interested reader is referred to the original paper [2]. In the EXTQF, using arithmetic codes to store extension bits naturally leads to a different rebuild method, discussed in Section 5.3.5.

5.2 Elements of a practical adaptive filter

In this section, we consider the components needed to develop a practical adaptive filter:

1. *The mechanism used to fix false positives.* When working with a single-hash function filter, this is equivalent to resolving a hash collision. In our discussion of the broom filter, we have already introduced fingerprint extensions. We can also resolve hash collisions through the use of *hash selectors* (5.2.1). These will be our two mechanisms of interest—selectors and extensions—and will be the basis of the practical adaptive filters we develop.

2. *The compression technique used to make the adaptivity mechanism succinct.* Arithmetic coding is optimal, allows the use of fractional bits per element, and can be sped up using approximation techniques. We discuss how to use arithmetic coding to represent selectors and extensions concisely and efficiently.

3. *Space reclamation heuristic.* An adaptive filter that allocates a fixed number of bits toward adaptivity must be able to reclaim adaptivity bits over time—otherwise, its ability to adapt will steadily degrade. Using arithmetic coding to store adaptivity information lends itself to a natural rebuilding mechanism.

4. *The remote representation architecture.* The remote representation R must support reverse lookups; i.e., R must provide a function $g : f(x) \mapsto x$, where $x \in S$ and $f(x)$ is its fingerprint. We develop a simple remote representation that is kept in sync with the local filter L.

5.2.1 Adaptivity mechanism

Perhaps the most important component of an adaptive filter is its ADAPT operation, which fixes a single false positive. For single-hash function filters, fixing a false positive is equivalent to preventing a hash collision from recurring. As we have seen, the broom filter fixes a false positive by extending its stored fingerprint until it no longer collides with a query's fingerprint. An alternative way to remove a hash collision is to store a different fingerprint entirely; this is the insight behind *hash selectors*.

Hash selectors

In the literature, the idea of using multiple hash functions to adapt first appears in Mitzenmacher's Adaptive Cuckoo Filter (ACF) [12]. Mitzenmacher's ACF fixes false positives by using multiple hash functions $h_0, h_1, ..., h_k$ where each h_i maps to p bits: $h_i : U \to \{1, ..., 2^p - 1\}$. Initially, all elements get their fingerprints from the same hash function h_0. When a hash collision occurs on a fingerprint $h_i(x)$, the ACF re-hashes x using the next hash function in the list, h_{i+1}, to generate a new fingerprint $h_{i+1}(x)$ which replaces $h_i(x)$ in the filter. This has the effect of resolving the hash collision with probability $1 - 1/2^p$ while maintaining x's membership in the filter.

Recall that to query an element's membership in a cuckoo filter (or any other single-hash-function filter), we hash the element and look for its hash in the filter. When using hash selectors, however, there are multiple hash functions to choose from, so it is important to know which hash function was used to generate each fingerprint in the filter. To deal with this issue, the ACF stores with each fingerprint an identifier marking which hash function that was used to generate the fingerprint. This identifier is the index of the hash function, i.e., i for h_i, and is called a hash selector.

To apply this technique to a quotient filter, we rehash remainders instead of rehashing whole fingerprints. The alternative is undesirable. Rehashing an element's whole fingerprint changes the element's quotient, meaning that the element's remainder must be shuttled to a new location in the quotient filter. Moving the remainder to a new location in the filter introduces a gap in the remainder's old run, which requires additional work to remedy. More importantly, the possibility that a query element has multiple quotients means that *all queries would need to check the runs of all possible quotients of each query element.* This would slow down queries considerably; a query would need to check all of the k possible fingerprints of a query element, increasing query time by a factor of k.

Hashing remainders and leaving quotients intact removes these problems from consideration. Each query element has only one location to check, and remainders can be quickly updated in-place. Because only the remainder is changed, however, this policy resolves false positives with lower probability: $1 - 1/2^r$ instead of $1 - 1/2^p$ (recall that r denotes remainder size, so $p > r$).

To identify the hash function used for a given fingerprint, we can store a hash selector with each remainder. Each hash selector can be stored succinctly using arithmetic coding.

5.2.2 Arithmetic coding

In this section, we discuss how to store extensions and selectors succinctly using arithmetic coding. In both cases, we store arithmetic codes at the block level. This means that an unencoded message

is a string of 64 selectors or extensions. Because our adaptive filters are based on the RSQF, which uses 2.125 metadata bits per element, we store a 56-bit arithmetic code for each block, bringing metadata space usage up to 3 bits per element.

Arithmetic coding for extensions

A fingerprint extension is a string of 0 or 1 bits. Length matters: the extension "0" is distinct from "00". To encode fingerprint extensions using arithmetic coding, we order all possible extensions by length and value (5.1) and group by length (5.2).

$$\varepsilon, 0, 1, 00, 01, 10, 11, 000, ..., 111, 0000, ... \tag{5.1}$$

$$(\varepsilon), (0, 1), (00, 01, 10, 11), (000, ..., 111), (0000, ..., 1111), ... \tag{5.2}$$

Then, to find the subinterval in $[L, H)$ corresponding to a particular extension x of length k, we first subdivide $[L, H)$ into intervals by length, where the subinterval for length l has length proportional to $\Pr(k = l)$. Next, because all extensions of a given length are equally likely, we split the subinterval for extensions of length k into 2^k evenly-sized pieces and take the piece corresponding to x's position in the sequence.

We then modify ENCODE to handle 64-element arrays of these extensions; the result is shown in Algorithm 5.2. It is worth unpacking line 9, reproduced below:

$$L \leftarrow L + r\Pr(k = 0) + r\Pr(k > 0)\Big(\sum_{i=1}^{k-1}\{2^{-i}\} + 2^{-2k} \cdot b\Big)$$

$$L \leftarrow L + r\Pr(k = 0) + \underbrace{r\Pr(k > 0)\sum_{i=1}^{k-1} 2^{-i}}_{\alpha} + \underbrace{r\Pr(k > 0) \cdot 2^{-2k} \cdot b}_{\beta}$$

This implements our two-step interval subdivision procedure. The α term accounts for the subintervals of $[L, H)$ corresponding to extension length. The β term accounts for the second-level subinterval that yields a particular extension.

Our implementation of a 56-bit arithmetic code can hold around 8 single-bit extensions or a single 19-bit extension.

Arithmetic coding for selectors

Storing selectors with arithmetic coding requires computing the probability that a selector takes on a particular value. The probability distribution is presented in analysis of the SELQF [11], and we reproduce a useful lemma below:

Lemma 5.2. Consider a sequence of $Q = x_1, x_2, \ldots, x_t$ of queries (interleaved with inserts), where each $x_i \notin S$ and Q consists of cn unique queries (with any number of repetitions), where $c < 1/\varepsilon - 1$.

```
1  function ENCODE(m)
2      L ← 0
3      H ← 1
4      for each a_i ∈ m do
5          r ← H − L
6          L ← L + r · Σ_{j=1}^{i−1} Pr(a_j)
7          H ← L + r · Pr(a_i)
8      return L
```

Algorithm 5.1: Encoding a message m with arithmetic coding.

```
1  function ENCODEEXTS(E)
2      L ← 0
3      H ← 2^64 − 1
4      for (b, k) ∈ E do
5          r ← H − L
6          if k = 0 then
7              H ← L + r Pr(k = 0)
8          else
9              L ← L + r Pr(k = 0) + r Pr(k > 0)(Σ_{i=1}^{k−1} {2^{−i}} + 2^{−2k} · b)
10             H ← L + r Pr(k > 0) · 2^{−2k}
11             if H − L ≤ 1 then
12                 return ENCODINGFAILURE
13     return L
```

Algorithm 5.2: Encoding a 64-element array of extensions with arithmetic coding.

Let $y \in \mathcal{S}$ and $v(y)$ be the hash-selector value of y after the t queries. Then,

$$\Pr[v(y) = k] < \begin{cases} \frac{1}{e^{c\varepsilon}} & k = 0 \\ \varepsilon^k \sum_{i=1}^{k} \frac{c^i}{i!} & k \geq 1 \end{cases}$$

Using these bounds, we can assign approximate probabilities to each of the possible selectors within a given range—for simplicity, we limit selectors with a bound of 6. When a fingerprint with a selector value of 5 needs to be fixed, we simply roll the selector value back to 0.

Optimized arithmetic coding for selectors

In this section, we describe an encoding optimized for the distribution of hash selector values when $\varepsilon = 1/256$, using 0.875 bits of space per element on average. It seems likely that an optimized method similar to this could be created for other parameter settings.

The goal of our optimization is to approximate each probability in Lemma 5.2 as the sum of a small number of powers of two, each of which can be calculated with right shifts.

To encode, we iterate over all 64 numbers in our array. Let `high` and `low` be the high and low points of the current interval, respectively, and let `r = high - low`. We update `low` using the following `switch` statement, taking advantage of fall-through behavior. Each case corresponds to the probability of the previous character. For example, $\Pr[v(y) = 0] \approx 0.7808$ is approximated by $1/2 + 1/4 + 1/32 = 0.78125$.

```
switch (letter) {
  default:
    return 0;
  case 6:
    low += (range >> 19) + (range >> 20) + (range >> 23);
  case 5:
    low += (range >> 14) + (range >> 16);
  case 4:
    low += (range >> 10) + (range >> 11);
  case 3:
    low += (range >> 6) + (range >> 8);
  case 2:
    low += (range >> 3) + (range >> 4) + (range >> 7) + (range >> 9);
  case 1:
    low += (range >> 1) + (range >> 2) + (range >> 5);
  case 0: ;
}
```

Figure 5.2: Updating `low` in the optimized arithmetic code.

A nearly-identical switch statement works to update `high`—each case is decremented by 1, and there is a `break` separating the cases. After each character is coded, we fail if `high - low` < 2.

Decoding works similarly. We iterate 64 times, each time decoding a single character. To decode a character, we check to see which subinterval the arithmetic code fits into. This requires testing

the subintervals for 0, then 1, up to 6, in sequence. Because most hash selector values are low—in fact, most are 0—these guesses have a low average cost.

5.2.3 Space reclamation and rebuilds

Arithmetic coding lends itself to a natural rebuilding scheme: whenever an arithmetic code "overflows", i.e., requires more than 56 bits to represent, we zero out the code, effectively discarding adaptivity information for a block in the filter.

5.2.4 Remote representation

The remote oracle R of our practical adaptive quotient filters is an array that stores the elements of S in the same order as L. Similar to how the ACF uses a cuckoo hash table updated in lock-step with a cuckoo filter [12], R stores $x \in S$ at the same index that L stores $f(x)$:

$$x \in S, f(x) = L[i] \implies x = R[i].$$

Looking up the element x associated with a fingerprint $f(x)$ is easy—we simply check the slot $R[s]$ where $\text{rem}(x) = R[s]$. This is convenient, because all fingerprint-to-element lookups are performed in a state where the location of the fingerprint in L is already known.

This approach to R is simple and easy to implement. The downside is that R must be updated whenever remainders shift positions in L, which happens reasonably often given how the RSQF, and quotient filters generally, shift remainders forward to make space during an insertion. When the local filter has high occupancy and insertions are likely to require several remainder shifts, shifting elements in R to maintain the correspondence between R and L can be expensive.

5.3 Extension-based adaptive filter

We now introduce the EXTQF, which translates the broom filter's theoretical design into a practical and performant filter. The EXTQF is partitioned into local and remote states L and R, where L is an augmented RSQF and R is an array updated in lock-step with L. Using an RSQF as the basis for L gives us fast insertion and query times. To represent extensions more succinctly, we use arithmetic coding. This minimizes the impact of storing extra data on filter performance and allows us to use fractional metadata bits per element, bringing L's metadata space usage per element from the RSQF's 2.125 bits to 3 bits.

5.3.1 Architecture

The local filter L is a blocked RSQF augmented to store an arithmetic code representing extension bits. More specifically, each block in L stores an arithmetic code representing 64 extensions, one for each slot in the block (Figure 5.3). We call this code an *extension code* and dedicate 56 bits in the block to storing it, bringing the metadata used per element to 3 bits.

The EXTQF's remote representation R is a single large array updated in correspondence with L as described in Section 5.2.4.

occupieds (64 bits)	
runends (64 bits)	
remainders ($64 \times r$ bits)	
offset (8 bits)	extension code (56 bits)

Figure 5.3: A block in the Extqf's local filter L.

5.3.2 Insertions

Inserting an element x into an Extqf $Q = (L, R)$ is detailed below (Algorithm 5.3). Effectively, an insertion is the same for an Extqf as it is for an Rsqf, with the addition of updates to R (lines 7, 15, and 24). Inserting x into R is simple (lines 7 and 24). When L shifts remainders and their associated metadata to make room for $f(x)$, we shift elements in R using ShiftR to maintain the correspondence between R and L. ShiftRemsAndMetadata is altered from the Rsqf, as it must also support extensions. This yields the new function ShiftRemsAndMetadataWithExts, which augments ShiftRemsAndMetadata to handle the decoding, modification, and re-encoding of extension codes.

Note that the Extqf does not adapt on insertions, unlike the broom filter. The most common access patterns for filters in LSM-tree-based key-value stores is a sequence of insertions (a "load phase") followed by a sequence of queries interlaced with a small minority of insertions. To optimize for this workload, we leave extensions untouched in insertions, deferring the cost of adapting to the second sequence.

5.3.3 Queries

Querying the membership of an element x in an Extqf requires checking each fingerprint's extension bits in addition to its remainder. These changes are reflected in lines 9–10 of Algorithm 5.4.

After using Rank and Select to find the runend corresponding to $\text{quot}(x)$, we walk through the run until we find a slot whose remainder matches the query's: $\text{rem}(y) = \text{rem}(x)$. In the Rsqf, Lookup would end here. In the Extqf, we must check $f(y)$'s extension (line 9). If the extension does not match, i.e., $\big(\text{quot}(y) \circ \text{rem}(y) \circ \text{ext}(y)\big) \not\sqsubseteq h(x)$, we continue walking through the run associated with the quotient $\text{quot}(x)$, terminating with absent if we fail to find any fingerprints whose remainders and extensions match $f(x)$'s in the run. If the extension does match, then we check if $y = x$ by using R to retrieve y from $f(y)$. Subsequently, the Extqf Adapts on x depending on whether the match constitutes a true positive or a false positive. In either case, the Extqf reports present, as the result of the access to R (line 10) should have no bearing on the output of the filter.[1]

[1] Otherwise, the filter would function as a fancy array.

```
1  function Insert(L, R, x)
2      p ← Rank(L.ρ, Select(L.θ, quot(x)))
3      if p < quot(x) then
4          L.θ[quot(x)] ← 1
5          L.ρ[quot(x)] ← 1
6          L.σ[quot(x)] ← rem(x)
7          R[quot(x)] ← x
8      else
9          u ← FirstUnused(L, p + 1)
10         if u = none then
11             u ← |L|
12             AddBlock(L)
13         UpdateOffsets(L, p + 1, u − 1)
14         ShiftRemsAndMetadataWithExts(L, p + 1, u − 1)
15         ShiftRemote(L, p + 1, u − 1)
16         if L.θ[quot(x)] = 1 then
17             UpdateOffsets(L, p)
18             L.ρ[p] = 0
19         else
20             UpdateOffsetsNewrun(L, quot(x), p)
21             L.θ[quot(x)] ← 1
22         L.ρ[p + 1] ← 1
23         L.σ[p + 1] ← rem(x)
24         R[p + 1] ← x
```

Algorithm 5.3: Insertion algorithm for the ExtQF.

```
1  function Lookup(L, R, x)
2      if L.θ[quot(x)] = 1 then
3          i ← Rank(L.ρ, Select(L.θ, quot(x)))
4          if i < quot(x) then
5              return ABSENT
6          else
7              repeat
8                  if rem(x) = L.σ[i] then
9                      e ← DecodeExt(L, i)
10                     f ← quot(x) ∘ rem(x) ∘ e
11                     if f ⊑ h(x) then
12                         if x ≠ R[i] then Adapt(L, R, x, i)
13                         return PRESENT
14                 i ← i − 1
15             until i < quot(x) or L.ρ[i] = 1
16     return ABSENT
```

Algorithm 5.4: Query algorithm for the Extqf.

5.3.4 Adapts

This brings us to the Extqf's Adapt algorithm (Algorithm 5.5). Like the broom filter, the Extqf adapts by extending fingerprints. Unlike the broom filter, the Extqf places extensions in a block-wise extension code. Therefore, to update an extension in the Extqf, we must decode the extension code of the block holding the extension; modify the decoded array of extensions; and re-encode the updated array, storing the resulting value as the new extension code of the block.

In our implementation, we take care to decode only when we need to. When we need to retrieve the extension bits of multiple elements from the same block, we only decode the block's extension code once. In parts of the following pseudocode, however, we present a simplified view of the decoding process that, if taken at face value, would result in multiple extraneous decodes. In the pseudocode, this simplified view is represented by DecodeOne(L, i), a function that returns the extension bits for a single slot i in L. When we find it necessary to refer to encoding at the block level, we use DecodeBlock(L, j), a function that returns a 64-element array of the extension bits for all slots in the j-th block in L. DecodeOne is defined in terms of DecodeBlock. EncodeBlock corresponds to DecodeBlock.

Because the Extqf does not adapt on insertions, it is possible for the Extqf to end up storing duplicate fingerprints, each associated with different elements in S. This occurs when S contains two distinct elements $x, y \in S$ with conflicting hashes, $\text{pre}(h(x), p) = \text{pre}(h(y), p)$ (Recall that $p = \log(n/\varepsilon)$ is the length of the Extqf's baseline fingerprint).

Therefore, when a potential false positive is encountered for a query element q for the fingerprint $f(a)$ where $f(q) = f(a)$, we must first check the rest of the run containing $f(a)$ to ensure that there is not another fingerprint in the run corresponding to q (lines 2–6). Once we have confirmed that q is a false positive, we adapt on all elements that have fingerprints colliding with $f(q)$ (lines 7–13).

```
function DecodeOne(L, i)
    E ←DecodeBlock(L, i/64)
    e ← E[i mod 64]
    return e

function DecodeBlock(L, i)
    c ← L[i/64].extcode
    E ←Decode(extcode, 64)
    return E
```

Figure 5.4: Defining DecodeExt and DecodeBlock in terms of Extqf metadata and Decode.

```
 1 function Adapt(L, R, x, i)
 2     j ← i − 1
 3     repeat
 4         if R[j] = x then return
 5         j ← j − 1
 6     until j < quot(x) or L.ρ[j] = 1
 7     j ← i − 1
 8     repeat
 9         e ← DecodeOne(L, i)
10         if L.σ[i] = rem(x) and (quot(x) ∘ rem(x) ∘ e) ⊑ h(x) then
11             y ← R[i]
12             AdaptOnce(L, R, i, x, y)
13     until j < quot(x) or L.ρ[j] = 1
```

Algorithm 5.5: Adapt over a run in the Extqf.

```
 1 function AdaptOnce(L, R, i, x, y)
 2     E ← DecodeBlock(L, i/64)
 3     e′ ← ShortestDiffPre(h(y), h(x))    ▷ Takes the shortest prefix of h(y) that
                                           differs from h(x)
 4     E[i mod 64] ← e′
 5     c ← EncodeBlock(E)
 6     if c = EncodingFailure then         ▷ Failure occurs when the extension code
                                           overflows
 7         E ← [ε, ..., ε]
 8         E[i mod 64] ← e′
 9         c ← EncodeBlock(E)
10         if c = EncodingFailure then
11             E[i mod 64] ← ε
12             c ← EncodeBlock(E)
13     L[i/64].extcode ← c
```

Algorithm 5.6: Adapt on a single location in the Extqf. ε is the empty extension.

5.3.5 Rebuilds

When fixing a single false positive, ExtQF decodes the appropriate block's extension code, updates an extension in the decoded array, and attempts to re-encode the result. Note that the length of the arithmetic code representing the block's extension bits will scale with the number of extension bits. When the arithmetic code grows past 56 bits, it is no longer possible to store the arithmetic code in the block: the encoding fails (line 6). To resolve this issue, the ExtQF "rebuilds" the extensions in the block, i.e., it sets all extensions to the empty extension, ε (line 7).

In an attempt to fix the false positive that caused the rebuild, the ExtQF tries to update the extension that caused the overflow, leaving all other extensions empty (line 9). If this fails (line 10), the ExtQF clears this extension as well. Finally, the ExtQF encodes the resulting array of extensions and places the result in the appropriate block.

Rebuilds are problematic for the ExtQF because they represent a loss of information: by rebuilding a block, the ExtQF forgets about all of the false positives that it previously fixed. On a fundamental level, this problem is not unique to the ExtQF—it is unavoidable for any adaptive filter that has finite space. Without consuming increasing amounts of space, an adaptive filter cannot remember all past false positives. It can only remember some of them; the rest must be discarded.

How often an adaptive filter rebuilds determines how many unique queries it can handle before its performance starts degrading. If rebuilds occur often, then the filter will do no better than the unaugmented filter from which its local state is built (the RSQF, in the ExtQF's case).

5.4 Selector-based adaptive filter

In this section, we introduce the selector-based adaptive quotient filter (SelQF), which rehashes remainders to resolve false positives. We discuss the SelQF's architecture and core operations.

5.4.1 Architecture

The SelQF shares most of its structure with the ExtQF, the primary difference being that the SelQF stores hash selectors instead of fingerprint extensions.

A SelQF (L, R) consists of a local RSQF L and a remote state R that, like the ExtQF, is a large array containing the elements of S ordered to match the fingerprints in L. Like the ExtQF, the RSQF stores its adaptivity information (hash selectors) in an arithmetic code of 56 bits stored at the block level, bringing up the RSQF's metadata usage to 3 bits per element. In line with the naming of extension codes, this arithmetic code is called a *selector code*.

The SelQF's insertions and queries closely resemble the ExtQF's. SelQF insertions are almost identical to ExtQF insertions. SelQF queries, on the other hand, are slower than ExtQF queries because the SelQF cannot defer selector code decoding like the ExtQF can.

5.4.2 Insertions

Elements are hashed with h_0 at insertion time; equivalently, new elements are inserted with 0-valued selectors. To simplify insertions, each block's selectors are set to 0 by default; this codes to a 0-valued selector code for each block (see Section 5.2.2). This enables us to avoid performing any arithmetic encoding or decoding operations when inserting a new element into an empty slot.

occupieds (64 bits)	
runends (64 bits)	
remainders ($64 \times r$ bits)	
offset (8 bits)	selector code (56 bits)

Figure 5.5: A block in the SelQF.

Like the ExtQF, the SelQF does not adapt at insertion time. This has two important effects. First, because there are no Adapt operations performed during insertions and all selectors start at 0, a filter will have no nonzero hash selectors until the first time the filter is queried. This means that a sequence of insertions to an empty filter will not require any arithmetic coding operations, as selectors never need to be decoded. This speeds up insertions on a workload that consists of a sequence of insertions (e.g., a "load phase") followed by a sequence of queries, which is fairly typical.

Second, insertions do not increase the likelihood of rebuilds. Rebuilds happen when a block's selector code overflows, so they grow more likely as a block's selector code fills up. Not adapting on inserts means that selectors are not incremented on inserts, so a freshly initialized SelQF will not have any nonzero selectors, giving it more time before its first rebuild.

When insertions are interleaved with queries, the SelQF must decode and encode selectors to ensure that nonzero selectors stay aligned with their associated remainders (Algorithm 5.7, line 16).

5.4.3 Queries

SelQF queries closely resemble RsQF queries: finding an element requires searching for its fingerprint in a run of remainders, using rank and select operations to quickly find the appropriate run. The main difference is that in the SelQF, a block's selectors must be decoded before its remainders can be compared to the query element's. This is because, for the SelQF, each element in U has multiple potential remainders. To determine which of a query element's remainders should be compared to a remainder stored at i, we must first retrieve the selector at i, which requires decoding the $\lfloor i/64 \rfloor$-th block's selector code.

This slows down SelQF queries relative to ExtQF queries. In SelQF queries, selector decoding is always performed first, because remainders are meaningless without their selectors. In contrast, the ExtQF's remainders are readily interpretable; remainders can be read directly and extensions only need to be decoded when a matching remainder has been found. When a matching remainder is not found, the ExtQF can get away with not performing any arithmetic coding operations.

5.4.4 Adapts

The SelQF fixes false positives by incrementing the hash selectors of the elements at fault. To illustrate, suppose that a false positive arises because of a query element $x \notin S$ and an element $y \in S$ stored with the selector s, so $\text{quot}(x) = \text{quot}(y)$ and $r_s(x) = r_s(y)$. The SelQF fixes this

```
1  function INSERT(L, R, x)
2      p ← RANK(L.ρ, SELECT(L.θ, quot(x)))
3      if p < quot(x) then
4          L.θ[quot(x)] ← 1
5          L.ρ[quot(x)] ← 1
6          L.σ[quot(x)] ← rem(x)
7          R[quot(x)] ← x
8      else
9          u ← FIRSTUNUSED(L, p + 1)
10         if u = NONE then
11             u ← |L|
12             ADDBLOCK(L)
13         UPDATEOFFSETS(L, p + 1, u − 1)
14         SHIFTREMSANDMETADATA(L, p + 1, u − 1)
15         SHIFTREMOTE(L, p + 1, u − 1)
16         SHIFTSELS(L, p + 1, u − 1)
17         if L.θ[quot(x)] = 1 then
18             UPDATEOFFSETS(L, p)
19             L.ρ[p] = 0
20         else
21             UPDATEOFFSETSNEWRUN(L, quot(x), p)
22             L.θ[quot(x)] ← 1
23         L.ρ[p + 1] ← 1
24         L.σ[p + 1] ← rem(x)
25         R[p + 1] ← x
```

Algorithm 5.7: Insertion algorithm for the SELQF.

```
1  function Lookup(L, R, x)
2      if L.θ[quot(x)] = 1 then
3          i ← Rank(L.ρ, Select(L.θ, quot(x)))
4          if i < quot(x) then
5              return ABSENT
6          else
7              repeat
8                  s ← DecodeSel(L, i)
9                  if rem(x, s) = L.σ[i] then
10                     if x ≠ R[i] then Adapt(L, R, x, i)
11                     return PRESENT
12                 i ← i − 1
13             until i < quot(x) or L.ρ[i] = 1
14     return ABSENT
```

Algorithm 5.8: Query algorithm for the SelQF.

false positive by incrementing s and rehashing the remainder, resulting in the new fingerprint $f(y)$, where

$$f(y) \equiv \text{quot}(y) \circ r_{s+1}(y) \circ (s+1).$$

This fixes the false positive with probability $1 - 1/2^r$, i.e., the probability that y's new remainder, $r_{s+1}(y)$, doesn't match x's, $r_{s+1}(x)$. Unlike the ExtQF, however, this only fixes a *single* false positive; the probability that an element $z \neq x$, $z \notin S$ collides with y is left unchanged by the SelQF's Adapt operation; it is still $1/2^p$. ExtQF adapts, in contrast, reduce the probability of *all* collisions to $1/2^{p+n}$, where n is the number of extension bits. In other words, the SelQF does not benefit from serendipitous corrections.

To rehash y's remainder as $r_{s+1}(y)$, the SelQF must retrieve s, the selector value associated with y, and the element y itself. To retrieve s, the SelQF decodes the selector code of the block holding $f(y)$ (Algorithm 5.9, line 9). To retrieve y, the SelQF references the remote representation R (Algorithm 5.10, line 12).

5.4.5 Rebuilds

Because selector codes pertain to particular blocks, rebuilds are also executed at the block level. To rebuild a block, the SelQF sets all selectors in the block to 0 and rehashes all of its stored remainders using h_0. Rebuilding is more expensive for the SelQF than for the ExtQF because the SelQF must rehash all elements in the block.

Experimental results suggest that the SelQF rebuilds less often than the ExtQF; in any case, the SelQF appears to retain information more efficiently than the ExtQF. For a full discussion of experimental results, see Chapter 6.

```
1  function Adapt(L, R, x, i)
2      j ← i − 1
3      repeat
4          if R[j] = x then return
5          j ← j − 1
6      until j < quot(x) or L.ρ[j] = 1
7      j ← i − 1
8      repeat
9          s ← DecodeOne(L, i)
10         if L.σ[i] = rem(x, s) then
11             AdaptOnce(L, R, i)
12     until j < quot(x) or L.ρ[j] = 1
```

Algorithm 5.9: Adapt over a run in the Selqf.

```
1  function AdaptOnce(L, R, i)
2      T ← DecodeBlock(L, i/64)
3      s ← T[i mod 64] + 1
4      T[i mod 64] ← s
5      if EncodeBlock(T) fails then
6          T ← [0, ..., 0]
7          T[i mod 64] ← s
8          if EncodeBlock(T) fails then
9              T[i mod 64] ← 0
10             EncodeBlock(T)
11     L[i/64].selcode ← c
12     L.σ[i] ← r_s(R[i])
```

Algorithm 5.10: Adapt on a single location in the Selqf.

5.5 Summary

The broom filter is an adaptive filter with strong theoretical guarantees. It is partitioned into two parts: a single hash function filter L and oracle R. L allows the broom filter to maintain filter-grade query times, while R provides information to make $B = (L, R)$ adaptive. The broom filter fixes a false positive by extending the faulty fingerprint in L. This resolves the hash collision that caused the false positive, with the added benefit of reducing the probability of future collisions.

Developing a practical adaptive filter requires consideration of the mechanism used to fix false positives; the compression technique used to make the adaptivity mechanism succinct; a space reclamation heuristic to allow on-demand adapts; and a reasonable remote representation.

The ExtQF is a practical implementation of the broom filter that uses arithmetic coding to store fingerprint extensions succinctly. The ExtQF's local state is a blocked RsQF. Extensions are tracked and stored at the block level as extension codes. The ExtQF's insertions and queries are similar to the RsQF's, with the addition of remote state maintenance and extension encoding/decoding operations. Unlike the broom filter, the ExtQF does not adapt on insertions.

The SelQF adapts by rehashing remainders to resolve hash collisions. Like the ExtQF, the SelQF inherits the fundamental structure of its insertions and queries from the RsQF. Selectors can be stored efficiently by using an approximate arithmetic code that replaces floating point operations with additions and bit shifts. SelQF insertions are expected to have similar performance to ExtQF insertions. SelQF queries, however, are expected to be slower, due to the SelQF's need to perform arithmetic coding operations whenever a queried element's quotient matches a stored element's quotient. The ExtQF, in contrast, only needs to perform arithmetic coding operations on queries that match a full baseline fingerprint.

Chapter 6

Experimental Results

In this chapter, we empirically evaluate the SELQF and EXTQF. Our evaluation measures two important criteria: accuracy and speed. We measure each filter's accuracy by examining its false positive rates on real and synthetic data sets. As part of our synthetic tests, we develop a simple variable-memory adversary to approximate each filter's worst-case performance.

To see whether the filters are practical, we measure their throughput performance on queries and insertions. Furthermore, to measure the time cost of adaptivity, we evaluate variants of our filters that are more or less adaptive.

6.1 Methodology

Our tests are split into two broad categories: *false positive rate* (FPR) tests and throughput tests. FPR tests measure how well a filter can fix false positives over time. Throughput tests measure how quickly a filter can perform insertions and queries. We compare the SELQF and EXTQF with the best filters in each class. In the FPR tests, we compare the EXTQF and SELQF with adaptive filters; in the throughput tests, we compare our filters against high-throughput static filters.

Specifically, in FPR tests, we compare our filters with the Cuckooing ACF [10], the Cyclic ACF (with $s = 1, 2, 3$ hash selector bits), and the Swapping ACF [12]. (See Chapter 2 for a detailed description of each filter.) The Cyclic and Cuckooing ACFs have bins of size 1 and use four random hash functions to choose the location of each element. The Swapping ACF uses bins of size 4 and two location hash functions. To provide a baseline, we also include a static cuckoo filter. In throughput tests, we compare the SELQF and EXTQF against the vacuum filter [20], as well as the Uncompressed SELQF (USELQF), a SELQF variant that does not perform arithmetic coding operations; selectors are stored using a full byte per element.

In both kinds of experiments, the procedure is fundamentally the same: we insert a set S into a filter Q and subsequently query Q using a query set A. The primary difference between the two tests is what we measure: FPR tests count the number of false positives encountered at query time as a ratio of the total number of queries $|A|$, and throughput tests measure the amount of time elapsed for insertions and queries.

All experiments are run on a workstation with Dual Intel Xeon gold 6240 18-core 2.6 Ghz

processors with 128G memory (DDR4 2666MHz ECC). All experiments were single-threaded.

6.1.1 Test parameters

A/S ratio

A useful quantity for determining the "difficulty" of an FPR test is the ratio A/S (shorthand for $|A|/|S|$), which measures the size of the query set relative to the size of the filter's membership set. Generally, a higher A/S value indicates a more difficult workload—in larger query sets, "fixed" false positives are separated by more interspersed queries.

For adaptive filters in particular, a larger query set means more false positives to remember. Each adaptive filter has a bound on how many false positives it can keep corrected, and exceeding this bound with a high A/S will lead the filter to begin forgetting and un-fixing previously fixed false positives. For adaptive filters that rebuild, as the SelQF and ExtQF do, a high A/S value will also hurt query and insertion throughput. Higher A/S values mean more false positives, which in turn means more adapts. Adapts increment selectors and lengthen extensions, respectively, leading the SelQF and ExtQF to spend more time rebuilding or performing arithmetic coding operations in addition to elevating the filter's FPR. Therefore, to get a better understanding of each filter's performance, we use varying A/S values.

Load factor

Another important test parameter is a filter's *load factor*: the proportion of slots occupied out of those available. For a quotient filter, higher load factors increase the frequency of longer runs, resulting in slower queries and insertions.

Filter size

While the *logical behavior* of a filter will not change irrespective of scale, the filter's throughput performance will vary with respect to how much memory it requires. A recurring theme in filter research is that smaller filters will fit higher in the memory hierarchy and benefit from faster accesses. Accordingly, filters with fewer elements are faster than filters with more elements, and an adaptive filter's remote representation will be faster when its elements are small and few. For this reason, we store 10^4–10^7 elements in each filter in our tests and keep remote representation sizes consistent by treating U as the set of 64-bit integers.

Fingerprint length

Fingerprint length is important insofar as it directly affects filter size. In all our experiments, we use 8-bit remainders for the ExtQF and SelQF. Because our quotient filters use 3 bits per element for adaptivity, they use a total of 11 bits per element. To test the other filters at the same size, we allocate 11 bits to the Swapping and Cuckooing ACFs, which require no extra metadata, and $11 - s$ bits to each of the Cyclic ACFs with s hash selector bits.

Data distribution

We source S and A from data sets, and the shape of our data matters. Adaptivity is most useful when the query distribution repeats itself; a flat distribution will fail to show adaptive filters at their

best, and is better served by a static filter. To measure this effect, we use several different data sets, specified in Section 6.2.

6.2 False positive rate tests

We measure FPR performance on three data sets: synthetic data from FireHose generators [1]; CAIDA network trace data [18]; and an "adversarial" distribution that simulates a simple, memory-bounded adversary.

6.2.1 FireHose streaming benchmarks

The FireHose Streaming Benchmarks suite has three stream generators. We use two of these generators: *active set* and *power-law*. Both generators output 64-bit integers.

Active set

The active set generator selects keys from a continuously evolving "active set" of keys. Each key in the active set is generated a limited number of times, then removed from the active set and replaced by a new key. The probability that a particular key is sampled varies in time according to a bell-shaped curve, creating a "trending" effect where a key appears occasionally, and then more frequently, before finally tapering off [1].

We generated 10 million 64-bit integers using the active set generator and split these elements across the query set A and member set S using the parameter A/S. We set `POW_EXP` to 0.5 in the active set generator to encourage query repetitions—each query is repeated approximately 57 times in the resulting data set.

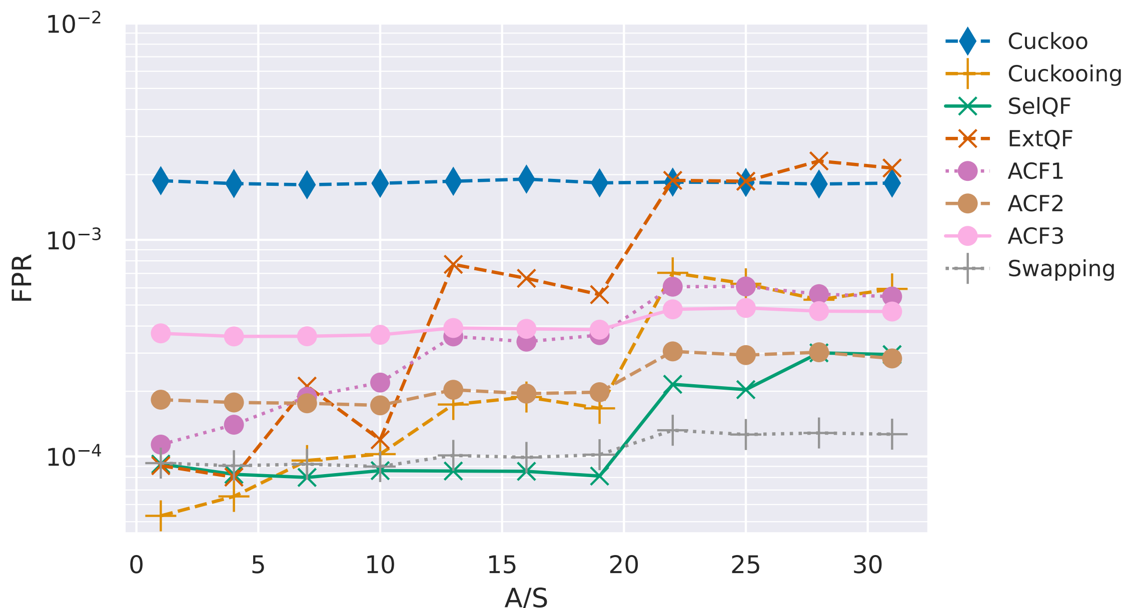


Figure 6.1: FPR on FireHose active set data, 10 million lines. Each element is repeated 57 times on average. Lower is better.

To test for various A/S values, we kept $|A|$ fixed (A was sourced from our generated data) and varied $|S|$. Our experimental results are presented in Figure 6.1. Each data point is the average of 10 iterations. ACF1, ACF2, and ACF3 represent the Cyclic ACF with $s = 1, 2, 3$ hash selector bits, respectively.

On active set data, the SelQF performs best at moderate A/S values in the range 5–20. Above $A/S \approx 20$, performance starts to degrade as the SelQF begins to perform more rebuilds, periodically discarding knowledge of past false positives. When A/S is in the range 20–35, SelQF FPR performance approaches ACF2 performance.

Power law

The power-law generator samples keys from a static range of 100,000 keys. The sampling is power-law distributed; keys at the low end of the range are generated much more frequently than keys at the high end [1].

We generated 50 million 64-bit integers using the power-law generator and queried them for multiple A/S values, using the same method as the active set experiments. Our results are presented in Figure 6.2. Each data point is the average of 10 iterations.

On power-law data, the SelQF performs best on moderate values, although this range is broader than it is for active set data.

On both data sets, the SelQF consistently outperforms the ExtQF. This indicates that the SelQF uses space toward adaptivity more effectively than the ExtQF. Moreover, serendipitous corrections appear to have little effect on FPR performance.

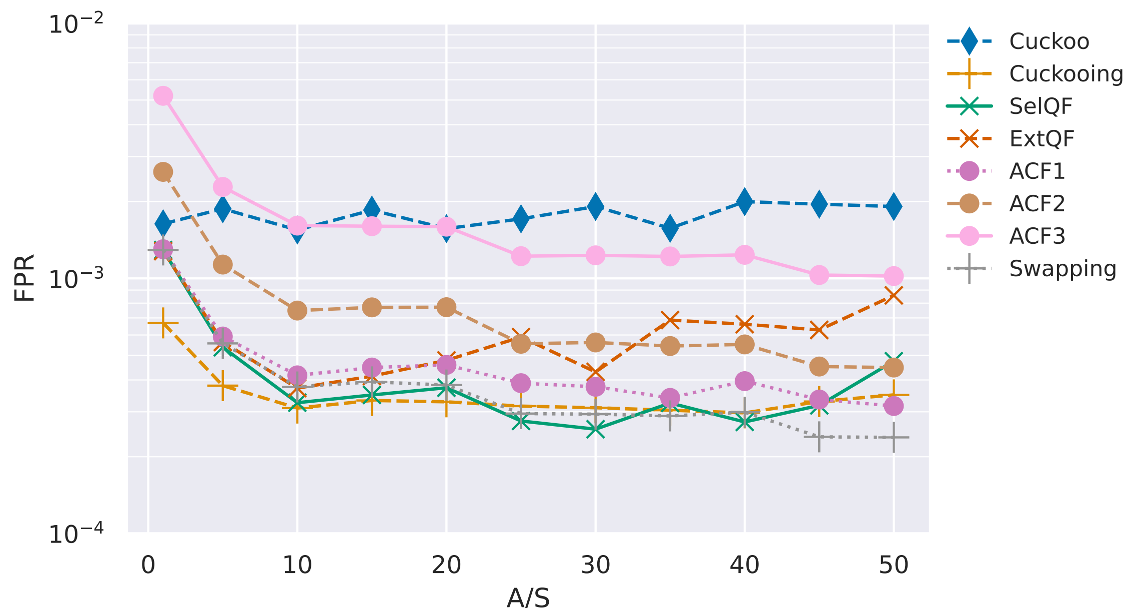


Figure 6.2: FPR on FireHose power-law data, 50 million lines. Each element is repeated 584 times on average. Lower is better.

6.2.2 Network traces

We replicate Mitzenmacher et al.'s experiments on real data [12] using the CAIDA 2014 internet traces data set [18], specifically the `equinix-chicago` and `equinix-sanjose` data sets. The results are presented in Figure 6.3.

On network trace data, all adaptive filters fix false positives successfully, so each adaptive filter's FPR performance is largely determined by its *baseline false positive rate*: the probability that a first-time query is a false positive. When s bits are used for adaptivity instead of for fingerprints, this increases the baseline FPR by 2^s. In Figure 6.3, this observation is borne out in the FPRs of the three Cyclic ACFs, which are stacked in a clear order. The ACF3, which uses the most bits toward adaptivity and has 3 fewer remainder bits, has the highest FPR of the three, followed by the ACF2. The SelQF and ExtQF, which use fractional bits per element for adaptivity, perform similarly to the Swapping ACF and the Cyclic ACF with $s = 1$. The Cuckooing ACF, which uses no explicit bits toward adaptivity, has the lowest baseline FPR.

6.2.3 Adversarial tests

The primary advantage of the SelQF and ExtQF is that they are adaptive in theory, even against an adversary [11]. To see how resilient the filters are to an adversary in practice, we test our filters against a simple adversary that remembers and queries previous false positives.

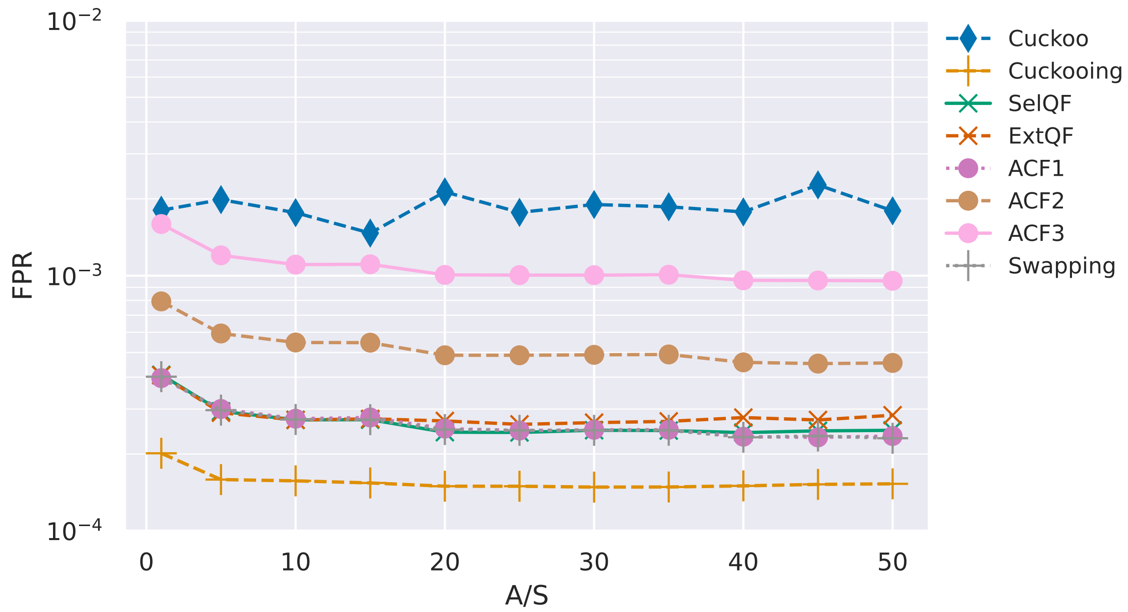
The adversary maintains a queue, Q, which is seeded with random queries. The filter being tested is also randomly seeded with a set S. The adversary queries the filter in multiple rounds, each divided into n subrounds. In each subround, the adversary queries the filter with each element in Q. After the n-th subround is complete, the adversary removes any elements in Q that were never false positives and advances to the next round. The adversary stops querying the filter after a fixed number of rounds.

The adversary wins when it finds a set of queries $Q' \subseteq Q$ where querying the filter with Q' across n subrounds leads the filter to err on each element of Q' at least once, thereby raising the filter's FPR to $1/n$ or above. For a filter to beat the adversary, it must continually reduce the number of elements in Q until $|Q|$ reaches 0.

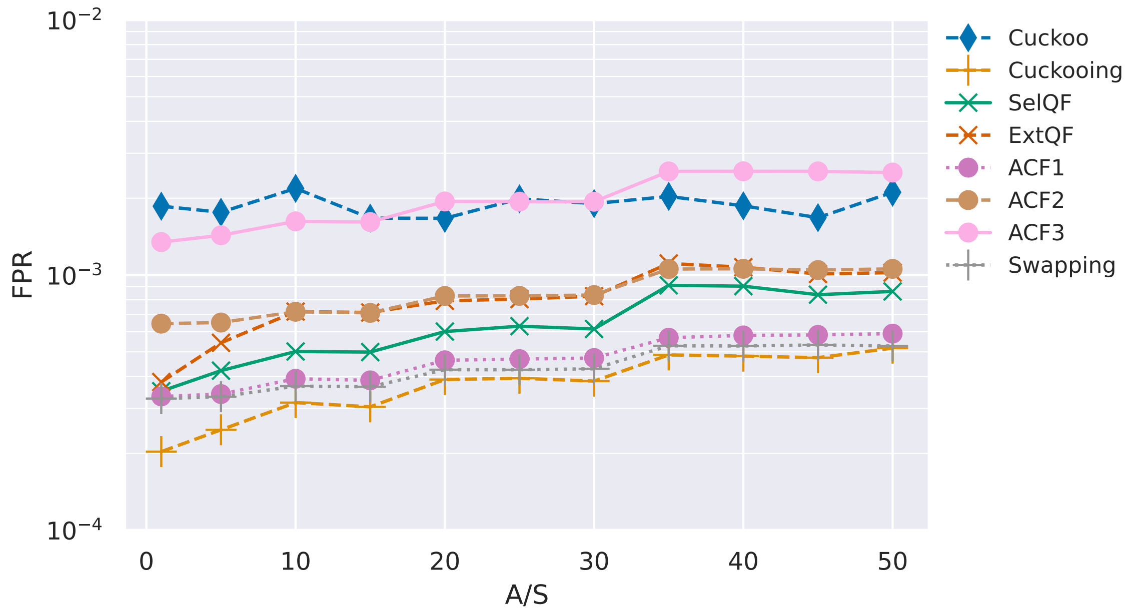
Figures 6.4 and 6.5 show the results of running the adversary with various initial Q/S values (shorthand for $|Q|/|S|$). The plot's x-axis is the round number; the y-axis is the size of Q at the start of each round. In both figures, the adversarial test was run for 10 iterations on each filter; each of these iterations is shown with a faded line, and their average is denoted with a dark line.

When initial Q/S is 10, only the SelQF and ACF3 beat the adversary, fully adapting by round 2. The other filters get stuck with larger Qs that remain at the same size round after round. As expected, the static Cuckoo filter shows the worst performance because it is not adaptive.

As the initial Q/S grows, the adversary has increasingly more elements to work with, effectively increasing the amount of memory that it has available. Any bounded adaptive filter is limited in how much it can adapt before it is forced to rebuild. With enough memory, the adversary can force the filter into *rebuild cycles*, wherein the same elements have their adaptivity information repeatedly grown, overflowed, and rebuilt. This can be seen in the failure of the SelQF and ExtQF against the adversary as initial Q/S grows past 10.

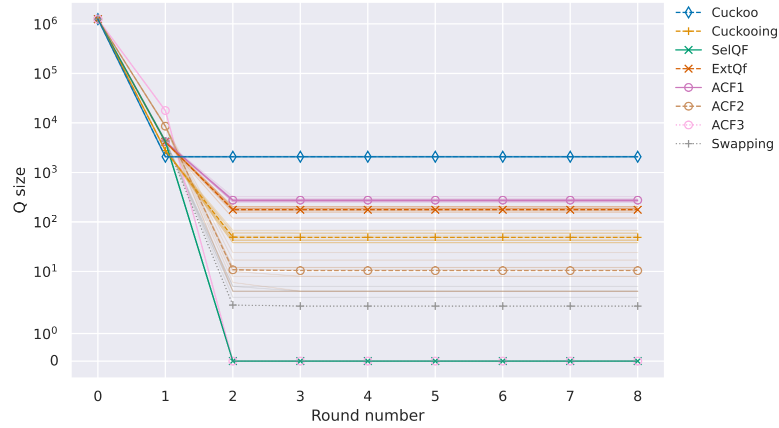


(a) FPR on `equinix-chicagoA`.

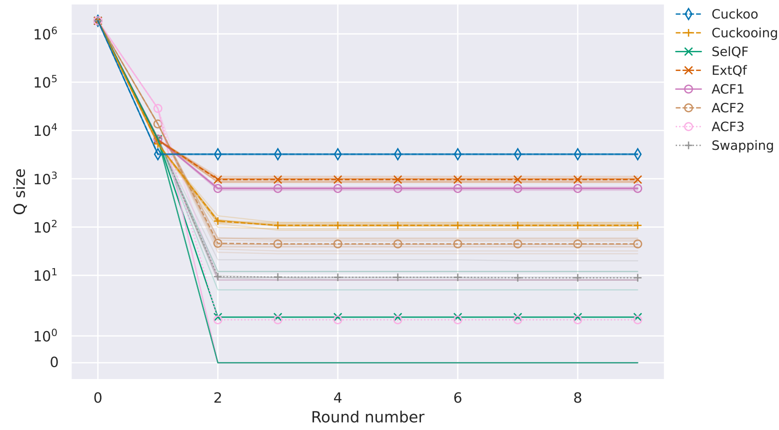


(b) FPR on `equinix-sanjose`.

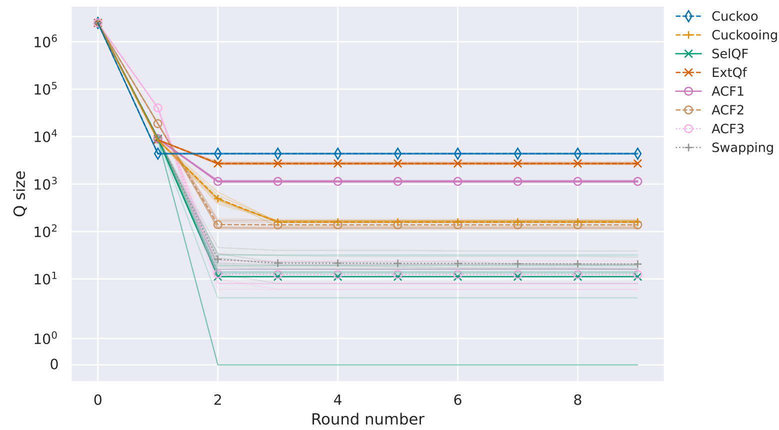
Figure 6.3: FPR results on CAIDA network trace data.



(a) Initial $Q/S = 10$. Only the SELQF and ACF1 adapt fully.

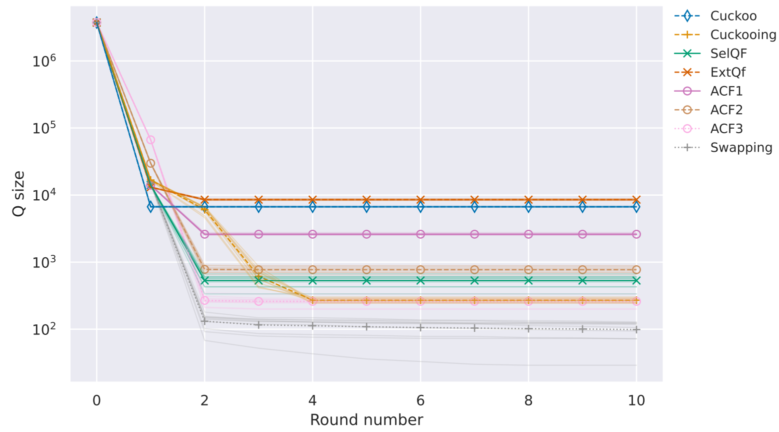


(b) Initial $Q/S = 15$. The SELQF and ACF1 adapts sometimes but not always.

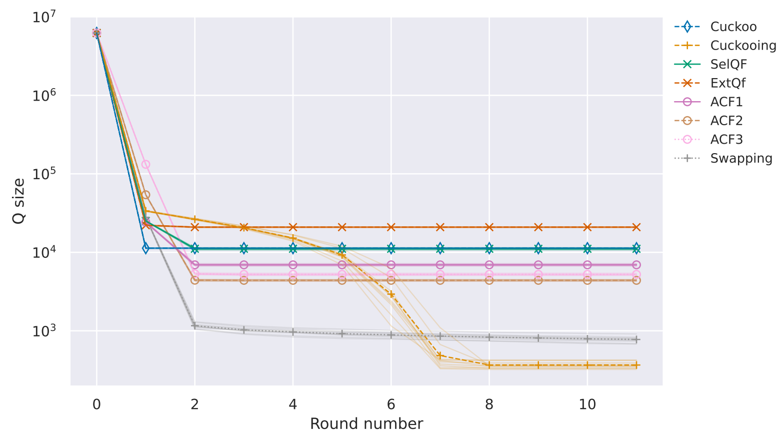


(c) Initial $Q/S = 20$. Only the SELQF adapts a few times.

Figure 6.4: Adversary runs with initial Q/S values 10, 15, and 20. Each test was run for 10 iterations on each filter. The faded lines represent single iteratoins; the dark lines represent the average of 10 iterations.



(a) Initial $Q/S = 30$. The Swapping ACF, Cuckooing ACF, and ACF3 outperform the SELQF. Only the Swapping ACF adapts fully at least once.



(b) Initial $Q/S = 50$. The Cuckooing and Swapping ACFs outperform all other filters. No filter fully adapts.

Figure 6.5: Adversary runs with initial Q/S values 30 and 50. Each test was run for 10 iterations on each filter. The faded lines represent single iteratoins; the dark lines represent the average of 10 iterations.

In our experimental results, we track the FPR for each round in two ways. The first is the *overall* FPR, which is the proportion of total false positives to total queries. The second is the *element-wise* FPR, which is the ratio of the number of elements in Q that had at least one false positive to the number of unique elements in Q. The overall FPR measures the filter's FPR on a given round of the adversary test. The element-wise FPR measures how effectively the filter gets rid of elements in Q.

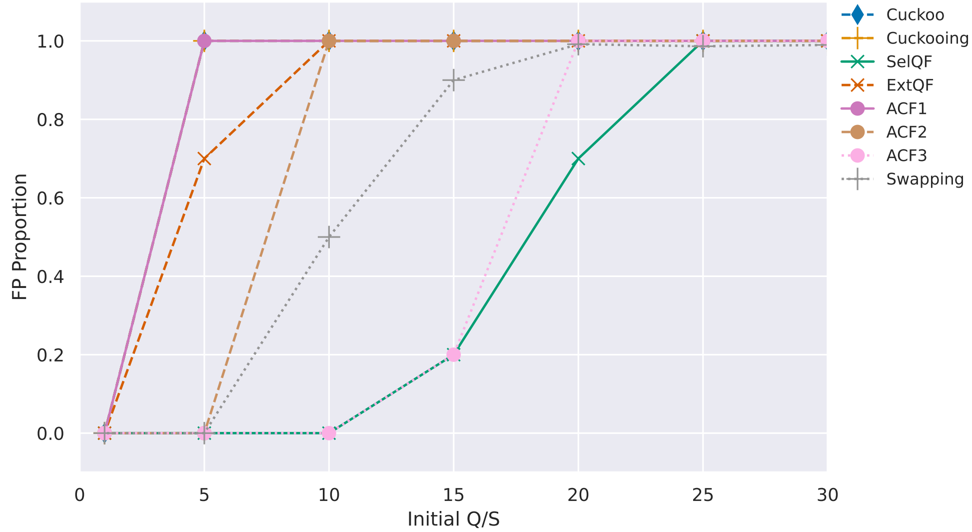


Figure 6.6: Adversary results summary. FP Proportion captures the proportion of all elements in Q that have at least one false positive.

We summarize our adversarial test results in Figure 6.6. The x-axis of our plot is the Q/S in the first round; the y-axis is the element-wise FPR in the last round of each test, averaged over 10 iterations. The SelQF does well up until around $Q/S \approx 20$, after which the adversary successfully forces false positives on all elements in Q. This agrees closely with the analysis of the SelQF [11].

6.3 Throughput tests

We test the SelQF and ExtQF against the vacuum filter, a cache-efficient modified cuckoo filter that uses a special fingerprinting function to improve locality [20]. We use the "from-scratch" version of the vacuum filter [19].

To isolate the cost of arithmetic coding, we also include the Uncompressed SelQF (USelQF), which performs no arithmetic coding operations and is allotted 8 hash selector bits per element. The USelQF is very space-inefficient.

We evaluated the throughput performance of each filter by inserting 2^{24} randomly-generated 64-bit integers into the filter before querying it with FireHose data, both active set and power-law. Our A/S values were implicitly determined by our data sets. We varied load factor to see its impact on throughput.

Our results for active set data are displayed in Figures 6.7 and 6.8; our results for power-law

data are shown in Figures 6.7 and 6.9. In each plot, the x-axis is the load factor; the y-axis is the number of operations performed per second. In the active set experiment, we reused the 10-million line data set from our FPR tests. In the power-law experiment, we used a 1-billion line data set. To account for noise, we ran each sequence of insertions and queries 10 times and averaged the results.

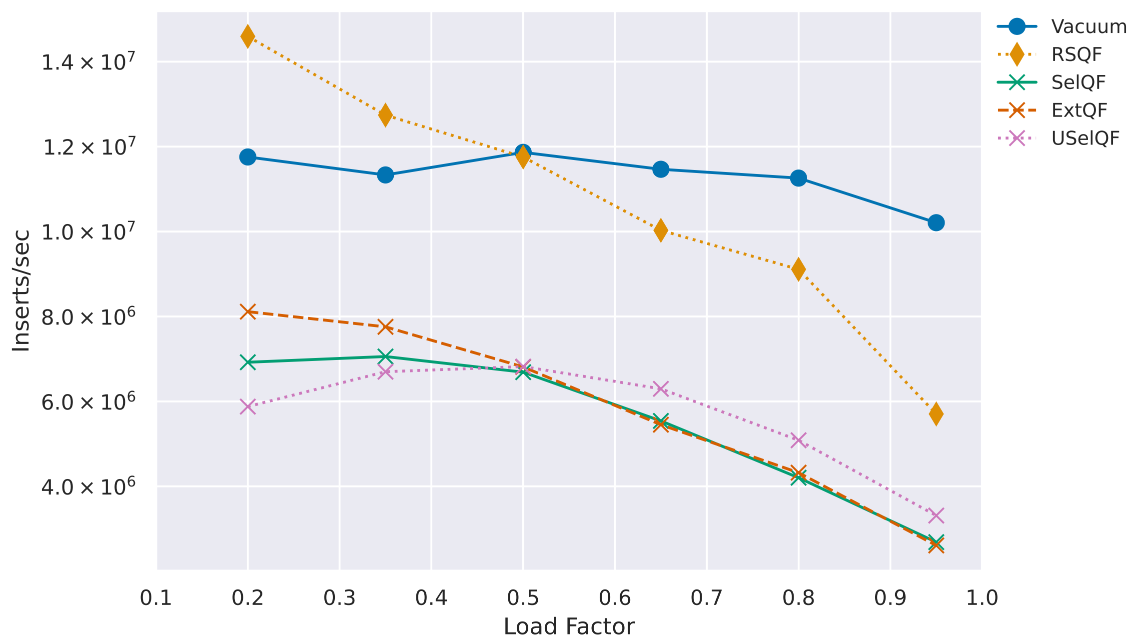


Figure 6.7: Insertion throughput over 2^{24} random insertions. Higher is better.

The SelQF and ExtQF attain high throughput on queries and insertions. The SelQF, however, has notably worse query throughput than the ExtQF, likely owing to the SelQF's need to decode selectors up-front, whereas the ExtQF can defer extension decoding to remainder matches only.

The vacuum filter's throughput performance is largely unaffected by load factor, unlike the RSQF, whose throughput declines as load factor increases. This is due to the two filters' differing internal structures. The RSQF, being a quotient filter, accumulates longer runs, which take more time to traverse, as load factor increases.

Interestingly, the SelQF and USelQF have roughly equivalent insertion throughput. The cost of arithmetic coding has a significant impact on query throughput, however. The SelQF attains roughly 70% of the USelQF's query throughput on power-law data, a notable decrease.

The throughput experiment results highlight an important trade-off between the SelQF and ExtQF. Lengthening remainders, as the ExtQF does, has better query throughput but worse adaptivity per bit. On the other hand, rehashing remainders, as the SelQF does, has worse query throughput but better adaptivity per bit.

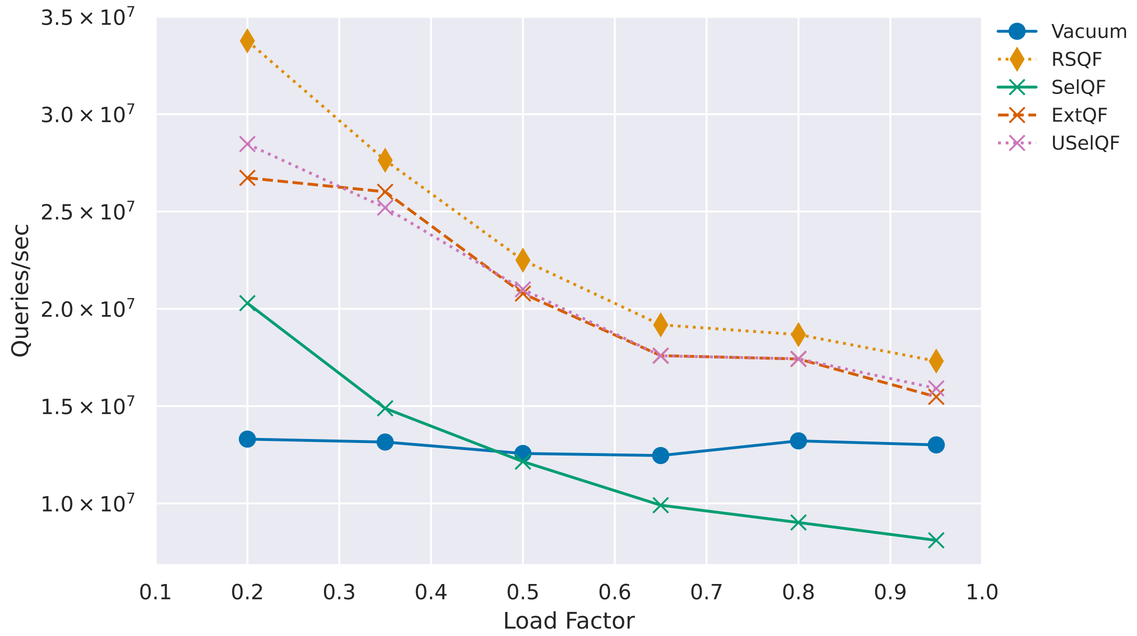


Figure 6.8: Query throughput on active set data, 10 million lines. Higher is better.

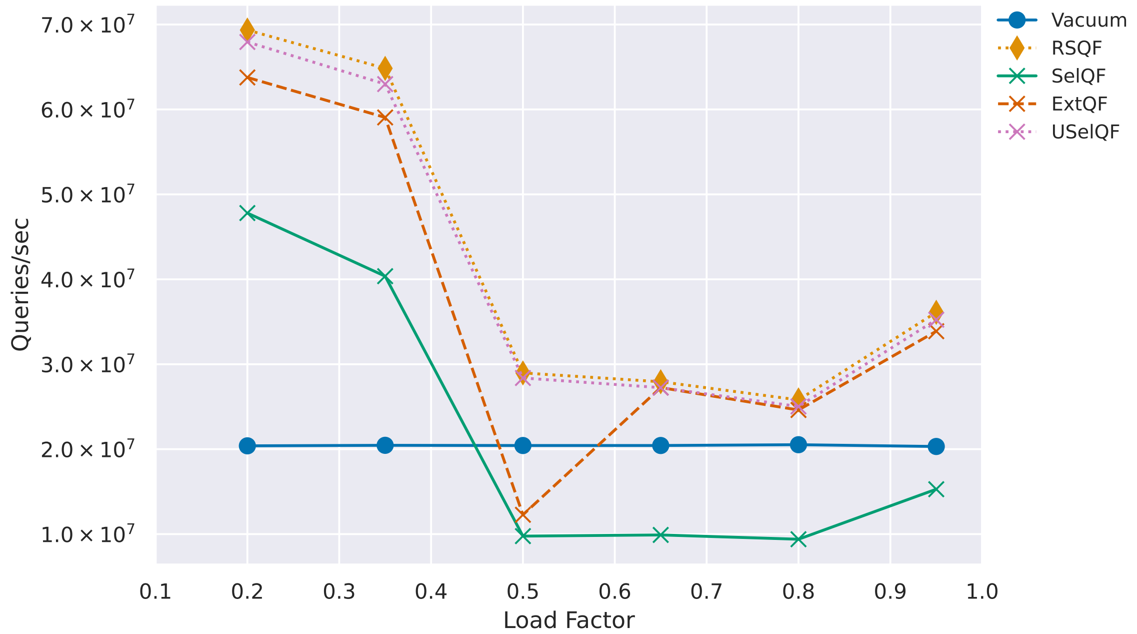


Figure 6.9: Query throughput results on power-law data, 1 billion lines. Higher is better.

6.4 Summary

Empirical evaluation of the SELQF and EXTQF shows that the SELQF matches or outperforms the FPR performance of other adaptive filters for moderate A/S values. This is in line with theoretical analysis [11]. The EXTQF does not perform as well as the SELQF; it appears to suffer from more frequent rebuilds, although additional testing is needed to ascertain the precise cause of the EXTQF's underperformance. Nonetheless, this experimental result indicates that hash selectors are a more efficient adaptivity mechanism than fingerprint extensions.

Adversarial tests also show that the SELQF matches or outperformes other adaptive filters on moderate A/S values. The adversary consistently wins for all filters on any A/S higher than 20; on A/S values under 10, meanwhile, it appears that all filters beat the adversary. In between, the SELQF has the lowest proportion of elements with at least one false positive. The EXTQF, on the other hand, loses to the adversary for any $A/S \geq 10$.

Throughput results indicate that the SELQF is within an order of magnitude of the vacuum filter for insertions. For queries, the performance gap is much narrower: at 95% load factor on power law data, SELQF queries are 75% as fast as vacuum filter queries. EXTQF queries, on the other hand, outperform vacuum filter queries, likely because the EXTQF can defer extension decoding to baseline fingerprint matches.

Chapter 7

Conclusion

This thesis has presented the Selqf and Extqf, two practical and provably adaptive quotient filters [2, 11] whose false positive performance matches or exceeds that of other state-of-the-art filters while maintaining high throughput. The use of arithmetic codes enables compact representation of adaptivity information, and our tailored implementations of arithmetic coding do not appear to dominate the Rsqf's overall throughput.

7.1 Review of results

The Selqf matches or outperforms the Extqf and other state-of-the-art adaptive filters on FPR tests. That the Selqf outperforms the Extqf indicates that hash selectors are a better use of adaptivity bits than fingerprint extensions. Determining the precise cause of the Extqf's underperformance requires additional work, although we suspect that frequent rebuilds are the primary cause.

The Extqf has better query throughput than the Selqf because the Selqf must decode all selectors up-front, whereas the Extqf does not need to decode extensions until a matching remainder is found. This shows a clear tradeoff between the two filters: the Selqf has better adaptivity properties, while the Extqf has higher query throughput.

7.2 Future directions

Looking forward, it would be helpful to analyze the behavior of the Selqf and Extqf on large-scale data. The quotient filter's primary advantage over the cuckoo filter is its cache efficiency, as insertions and queries access fingerprints that are stored in contiguous runs. This advantage is not readily apparent when test data is small.

Further analysis of the adaptive quotient filters' rebuild behavior is also needed. Rebuilds undoubtedly have a significant effect on filter performance, but detecting and mitigating them is difficult. As a first step, tracking rebuild behavior on FPR and adversarial tests could be fruitful.

Developing heuristics to preserve more data on rebuilds may also prove useful. Using alternative compression techniques to store adaptivity information or finding ways to restructure the Rsqf to implicitly store extra selector or extension bits may also help delay rebuilds, improving performance

on tests with lower A/S values.

So far as comparing filters is concerned, it appears that a more comprehensive study is needed. Results concerning filter performance do not appear easily reproducible—results from our tests, which used unmodified published code, often appeared to contradict paper findings. Having a better understanding of which parameters are most important for real-world workloads and performing a fair comparison of filters in the literature would help define the best use cases for the SelQF and ExtQF.

Bibliography

[1] Karl Anderson and Steve Plimpton. *FireHose streaming benchmarks*. Tech. rep. Sandia National Laboratories, 2015.

[2] Michael A Bender, Martin Farach-Colton, Mayank Goswami, Rob Johnson, Samuel McCauley, and Shikha Singh. "Bloom filters, adaptivity, and the dictionary problem". In: *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE. 2018, pp. 182–193.

[3] Michael A Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P Spillane, and Erez Zadok. "Don't thrash: how to cache your hash on flash". In: *Proc. VLDB Endowment* 5.11 (2012), pp. 1627–1637.

[4] Michael A. Bender, Rathish Das, Martin Farach-Colton, Tianchi Mo, David Tench, and Yung Ping Wang. "Mitigating False Positives in Filters: to Adapt or to Cache?" In: SIAM. 2021.

[5] Burton H. Bloom. "Space/Time Trade-Offs in Hash Coding with Allowable Errors". In: *Commun. ACM* 13.7 (July 1970), pp. 422–426. ISSN: 0001-0782. DOI: `10.1145/362686.362692`. URL: `https://doi.org/10.1145/362686.362692`.

[6] Andrei Broder and Michael Mitzenmacher. "Network applications of bloom filters: A survey". In: *Internet mathematics* 1.4 (2004), pp. 485–509.

[7] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. "Bigtable: A Distributed Storage System for Structured Data". In: *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2006, pp. 205–218.

[8] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. "Cuckoo Filter: Practically Better Than Bloom". In: *Proceedings of the 10th ACM International Conference on Emerging Networking Experiments and Technologies*. CoNEXT '14. Sydney, Australia: Association for Computing Machinery, 2014, pp. 75–88. ISBN: 9781450332798. DOI: `10.1145/2674005.2674994`. URL: `https://doi.org/10.1145/2674005.2674994`.

[9] Paul G. Howard and Jeffrey Scott Vitter. "Practical Implementations of Arithmetic Coding". In: *Image and Text Compression*. Ed. by James A. Storer. Boston, MA: Springer US, 1992, pp. 85–112. ISBN: 978-1-4615-3596-6. DOI: `10.1007/978-1-4615-3596-6_4`. URL: `https://doi.org/10.1007/978-1-4615-3596-6_4`.

[10] Tsvi Kopelowitz, Samuel McCauley, and Eli Porat. "Support optimality and adaptive cuckoo filters". To appear.

[11] David Lee, Samuel McCauley, Shikha Singh, and Max Stein. "Telescoping Filter: A Practical Adaptive Filter". In submission. 2021.

[12] Michael Mitzenmacher, Salvatore Pontarelli, and Pedro Reviriego. "Adaptive Cuckoo Filters". In: *Proc. 20th Workshop on Algorithm Engineering and Experiments (ALENEX)*. SIAM. 2018, pp. 36–47.

[13] Alistair Moffat, Radford M. Neal, and Ian H. Witten. "Arithmetic Coding Revisited". In: *ACM Trans. Inf. Syst.* 16.3 (July 1998), pp. 256–294. ISSN: 1046-8188. DOI: 10.1145/290159.290162. URL: https://doi.org/10.1145/290159.290162.

[14] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. "The Log-Structured Merge-Tree (LSM-Tree)". In: *Acta Inf.* 33.4 (June 1996), pp. 351–385. ISSN: 0001-5903. DOI: 10.1007/s002360050048. URL: https://doi.org/10.1007/s002360050048.

[15] Anna Pagh, Rasmus Pagh, and S Srinivasa Rao. "An Optimal Bloom Filter Replacement". In: *Proc. 16th Annual ACM-SIAM Symposium on Discrete Algorithms*. 2005, pp. 823–829.

[16] Prashant Pandey, Michael A. Bender, Rob Johnson, and Rob Patro. "A General-Purpose Counting Filter: Making Every Bit Count". In: *Proc. ACM SIGMOD International Conference on Management of Data*. 2017, pp. 775–787.

[17] Keith Schwarz. "Approximate Membership Queries". URL: http://web.stanford.edu/class/archive/cs/cs166/cs166.1196/lectures/14/Small14.pdf.

[18] "The CAIDA UCSD Anonymized Internet Traces". URL: http://www.caida.org/data/passive/passive_dataset.xml.

[19] Minmei Wang, Mingxun Zhou, Shouqian Shi, and Chen Qian. *Vacuum Filter*. 2019. URL: https://github.com/wuwuz/Vacuum-Filter.

[20] Minmei Wang, Mingxun Zhou, Shouqian Shi, and Chen Qian. "Vacuum Filters: More Space-Efficient and Faster Replacement for Bloom and Cuckoo Filters". In: *Proc. VLDB Endow.* 13.2 (Oct. 2019), pp. 197–210. ISSN: 2150-8097. DOI: 10.14778/3364324.3364333. URL: https://doi.org/10.14778/3364324.3364333.

[21] Ian H. Witten, Radford M. Neal, and John G. Cleary. "Arithmetic Coding for Data Compression". In: *Commun. ACM* 30.6 (June 1987), pp. 520–540. ISSN: 0001-0782. DOI: 10.1145/214762.214771. URL: https://doi.org/10.1145/214762.214771.