Master Thesis

Concurrent Dynamic Quotient Filters: Packing Fingerprints into Atomics

Robert Williger

Date: 29. March 2019

Supervisors: Prof. Dr. Peter Sanders
Tobias Maier

Institute of Theoretical Informatics, Algorithmics
Department of Informatics
Karlsruhe Institute of Technology

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen, als die angegebenen Quellen und Hilfsmittel benutzt, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet habe.

Karlsruhe, den 29. März 2019

Abstract

Quotient filters are approximate membership data structures that can be used for a variety of applications, from speeding up database accesses to uses in computational biology. We look at several improvements to common quotient filters which are orthogonal to each other and can be combined as needed. We show how to use compact arbitrary length data types to reduce the memory usage by up to seven times and get very close to the theoretical optimum. We also present two different approaches to building concurrent quotient filters that use localized locking or no locking at all. This leads to a good speedup with increasing thread counts and an increase in performance of up to four times compared to traditional locking techniques. Using the compact packing technique with concurrent quotient filters also has the benefit of reducing atomic operations and enabling lock elision optimizations. Additionally, we describe how to use a multilevel quotient filter structure to implement dynamic growing while still allowing a user defined maximum false positive rate and being less than 10% slower compared to non-growing variants in scenarios where growing is not necessary.

Zusammenfassung

Quotienten Filter sind probabilistische Datenstrukturen, die für verschiedene Verwendugszwecke eingesetzt werden können, von der Beschleunigung von Datenbankzugriffen bis zum Einsatz in der Bioinformatik. In dieser Arbeit befassen wir uns mit verschiedenen Ansätzen zur Verbesserung von Quotienten Filtern. Diese sind unabhängig voneinander und können frei miteinander kombiniert werden. Wir zeigen wie man Datentypen beliebiger Länge kompakt speichern kann um den Speicherverbrauch auf bis zu ein Siebtel zu senken. Damit können wir sehr nah an das theoretische Optimum gelangen. Wir stellen außerdem zwei verschiedene Heransgehensweisen zur Implementierung von parallelen Quotienten Filtern vor, die lokales Locking verwenden oder ohne Locks auskommen. Dadurch erreichen wire einen guten Speedup mit steigender Anzahl an Threads und eine bis zu vier fache Geschwindigkeitsverbesserung im Vergleich zu traditionellen Locking Ansätzen. In Kombination mit der kompakten Speicherweise können wir die Anzahl an nötigen atomaren Operationen senken und zusätzliche Locking Optimierungen umsetzten. Darüber hinaus beschreiben wir einen multilevel Quotienten Filter, der sowohl dynamisches Wachsen erlaubt, als auch eine benutzerdefinierte Obergrenze für die Wahrscheinlichkeit an falschen positiv-Antworten zulässt. Dabei ist diese Datenstruktur nur maximal 10% langsamer als nicht wachsende Varianten in Anwendugnsfällen bei denen wachsen nicht nötig ist.

Contents

Abstract v

1 Introduction 1
- 1.1 Motivation . . . 1
- 1.2 Overview . . . 1

2 Related Work 3

3 Preliminaries 7
- 3.1 Common Quotient Filters . . . 7
  - 3.1.1 Overview . . . 7
  - 3.1.2 Definition . . . 7
  - 3.1.3 Operations . . . 10

4 Concurrent Quotient Filters 15
- 4.1 Compact arbitrary length data types . . . 15
- 4.2 Simple Concurrency . . . 17
  - 4.2.1 Simple Locking Quotient Filters . . . 17
  - 4.2.2 Linear Probing Quotient Filters . . . 18
- 4.3 Advanced Concurrency . . . 18
  - 4.3.1 Locking Mechanism . . . 19
  - 4.3.2 Concurrent Operations . . . 19
- 4.4 Concurrent growing . . . 22
  - 4.4.1 Overview . . . 22
  - 4.4.2 Implementation . . . 23
- 4.5 Compact Concurrent Quotient Filters . . . 24

5 Fully Dynamic Quotient Filters 25
- 5.1 Overview . . . 25
  - 5.1.1 Overall Structure . . . 25
- 5.2 Level Structure . . . 26
  - 5.2.1 Level Size . . . 26
  - 5.2.2 Growing Levels . . . 27
- 5.3 Quick Insert . . . 27
  - 5.3.1 Concurrent Optimization . . . 28

5.3.2 Contains Optimization . . . 28
5.3.3 Analysis . . . 29

6 Experimental Evaluation **33**
6.1 Setup . . . 33
6.1.1 Environment . . . 33
6.1.2 Measurement . . . 33
6.2 Concurrency . . . 34
6.2.1 Speedup and Bloom Filter Comparison . . . 34
6.2.2 Fill Degree . . . 37
6.2.3 Influence of Fill Degree on Efficiency . . . 38
6.2.4 Remainder Bits . . . 39
6.2.5 Mixed Reads and Writes . . . 41
6.3 Growing and Dynamic Quotient Filters . . . 42
6.3.1 Overhead of Growing Quotient Filters . . . 42
6.3.2 Impact of Growing Operations . . . 43
6.3.3 Fill Degree . . . 44
6.3.4 Memory Usage . . . 46
6.3.5 Quick Insert . . . 48
6.3.6 Overhead of Hazard Pointers . . . 50

7 Conclusion **51**
7.1 Future Work . . . 51

Bibliography **53**

1 Introduction

1.1 Motivation

Quotient filters are approximate membership query data structures that are used in a variety of applications such as databases, networks and computational biology. For example we can speedup the lookup performance for keys in a database by using a quotient filter that also contains all keys. Any lookup operation first checks if the key is contained in the quotient filter and can abort if no key is found. Only when a key is present in the quotient filter do we actually have to access the database. This results in better overall performance.

Quotient filters represent a set of elements and can quickly answer membership queries, while only using a small amount of memory per inserted element. This means they can hold millions of elements while still easily fitting in main memory. Most filters can not perform dynamic growing and have one fixed sized for their whole lifetime. Additionally, aside from the very basic and slow Bloom filters, existing filters can either not be used in a concurrent setting with multiple threads at all or the parallelized filter variant does not scale to a large number of threads.

Throughout this work, we present different ways to overcome the aforementioned problems of quotient filters which results in a collection of quotient filter variants that can meet different user requirements in various scenarios.

1.2 Overview

The thesis is structured as follows: We start in section 2 with related work on the quotient filters and other approximate membership query data structures. In section 3 we describe the quotient filter data structure in general. In section 4 we present concurrent quotient filters and the underlying atomic arbitrary length data types. We describe the fully dynamic quotient filter in section 5. It uses a multilevel quotient filter structure that allows dynamic growing while still having the false positive guarantees of standard quotient filters. The evaluation and comparison of these different quotient filter variants is done in section 6. We summarize the results and give an outlook on future work in section 7.

2 Related Work

Approximate Membership Query (*AMQ*) data structures are used in a variety of applications such as databases, storage systems, networks, and computational biology [12, 14, 1]. They represent a set or multiset of elements and are used to test whether an element is a member of that set. A membership test on a quotient filter will either return that the element is not in the set or that the element is probably in the set. There is a small probability ε, the false positive rate, that the membership test shows that an element is present when in fact it was not inserted into the set. The accuracy loss of false positives comes at the benefit of better space-efficiency than complete dictionaries.

Bloom Filter The classic example of an approximate membership query data structure is the Bloom filter [4] which uses a bit array of size m where all bits are initially set to zero. When inserting an element into the Bloom filter, it is hashed with k different hash functions each mapping the element to a uniformly random position in the bit array. The bits at these k positions are set to one. To test the membership of an element the same k positions are obtained through the given hash functions. If not all of the bits at these positions are set to one then the element has not been previously inserted. If all bits are set to one then the Bloom filter reports that the element is present. This happens either if the element has really been inserted before or if we encounter a false positive, which happens with probability ε. Typically, k is a constant which means that both the insert and the contains operations run in constant time, independent of the size of the Bloom filter or the number of elements already in the set. The false positive rate of a Bloom filter depends on k and on the ratio of set bits to unset bits [15]. The disadvantage of a Bloom filter is that it can not delete elements once inserted and it is unable to grow dynamically (an upper bound to the number of elements has to be known during the initialization).

An improvement on the classical Bloom filter is the scalable Bloom filter described in [2]. There, the problem of dynamically growing a Bloom filter is addressed while also assuring an upper bound on the total false positive rate. This is done by using a series of classical Bloom filters of increasing size and decreasing false positive rate where new filters are added as needed. An insert or contains operation has to go through all of the present Bloom filters making it slower than the classical Bloom filter.

Another variant on the Bloom filter is the counting Bloom filter [7]. Given an upper bound to the number of insertions of one element, it supports deletions but it uses about 3 - 4 times more space than a classical Bloom filter for the same false positive rate.

Cuckoo Filter Another AMQ data structure besides the Bloom filter is the cuckoo filter [6]. Cuckoo filters use an array of buckets where each bucket can hold multiple fingerprints. Cuckoo filters work similarly to a cuckoo hash table. An insert operation uses two hash functions which generate the fingerprint and two bucket indices. If one of the buckets has an empty slot then the element is inserted into it. Otherwise, an element of one of the buckets is moved to its alternate location in another bucket in order to make a slot available for the new element. The average insertion time depends on the fill degree of the filter. To test the membership of an element, the two corresponding buckets are simply checked for the fingerprint of that element which is a constant time operation. Cuckoo filters also support the removal of elements. The space efficiency for a cuckoo filter is better compared to a Bloom filter for small false positive rates ($\varepsilon < 3\%$).

Variants on the cuckoo filter include the adaptive cuckoo filter [11] which uses an additional cuckoo hash table in order to remove false positive from future lookups. This can reduce the false positive rate significantly in certain scenarios.

Quotient Filters are AMQ data structures introduced by Bender et al. [3]. It is the basis of this thesis and is described in detail in section 3. Quotient filters have an array of integers where each integer stores one entry consisting of a fingerprint and 3 additional bits of status information. Insert operations work similar to linear probing which results in continuous clusters of elements inside the array. Besides insert and contains operations a quotient filter also supports the removal of elements, bounded dynamic growing, and merging of two quotient filters without rehashing in linear time. In order to count inserted elements with a quotient filter one can simply store multiple copies of the same fingerprint in the quotient filter but this can slow down the performance of operations compared to other AMQs which are specialized for counting the copies of elements.

A variation on basic quotient filters are counting quotient filters [13] which allow to count the number of occurrences of each input element. Bender et al. also present a different way of storing the status information of the entries in the quotient filter which can be leveraged by using rank and select bit-vector operations to speedup lookups at higher load factors, while also reducing the average memory needed to store the status information from 3 bits to 2.125 bits.

Bloom filters are easily parallelizable since they have multiple independent accesses at random locations in the bit array. This also results in bad cache efficiency. Cuckoo filters and quotient filters on the other hand, access elements more locally which makes them cache friendly but also more difficult to parallelize in a scalable way.

Hash Tables provide a similar functionality to AMQs but don't have any false positives for lookup operations. This comes at the cost of needing to store the entire element instead of a small fingerprint. There are however space efficient hash tables [8, 10] which provide fast access performance and only have a small memory overhead in addition to the storage space needed for storing the elements.

Robin Hood Hashing [5] is a hash table which uses linear probing but also orders elements similar to a quotient filter. The difference is that during the linear probing of an insertion, Robin Hood Hashing replaces the current element with the newly added element if its probe count is larger than that of the new element. It then continues the insertion with the replaced element with the same technique until an empty slot is found resulting in the exact order as a quotient filter where the hashed key is used as a fingerprint.

3 Preliminaries

3.1 Common Quotient Filters

3.1.1 Overview

The quotient filter (*QF*) as introduced in [3] is an approximate membership query data structure (AMQ) which allows insert and contains operations. For an element which was previously inserted into the quotient filter the contains operation will always return true. An element which has not been previously inserted into the quotient filter can still be found and declared present by the contains operation with a probability of ε which is called the *false positive rate*. A quotient filter can be dynamically resized but suffers from an increasing false positive rate as more elements are inserted.

3.1.2 Definition

A quotient filter contains a set of all fingerprints of elements $S \subseteq E$ where E is the universe of all possible elements. The p-bit fingerprint f of an element $e \in E$ is obtained by applying a hash function $h : E \to \{0, \ldots, 2^{64} - 1\}$ and then taking the p least significant bits: $f(e) = h(e) \mod 2^p$. An insert operation takes an element e and adds its fingerprint $f(e)$ to S. A contains operation takes an element e and checks whether the fingerprint $f(e)$ is contained in S. The p-bit fingerprint can be partitioned into a q-bit *quotient* and a r-bit *remainder* such that $p = q + r$. The quotient of a fingerprint f is defined as $f_q = \lfloor f/2^r \rfloor$ and the

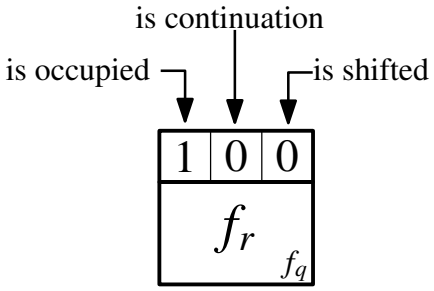


Figure 3.1: Depiction of a single slot. The three status bits are represented by 0 or 1 in the top row whereas the remainder f_r and the quotient f_q of the entry are found in the box below. Note that the quotient is only stored implicitly through the position of the entry in the array.

remainder as $f_r = f \mod 2^r$ and we use these notations throughout this thesis. For any given quotient f_q and remainder f_r the original fingerprint can be reconstructed uniquely as $f = f_q \cdot 2^r + f_r$ which can be computed quickly using bit operations.

The quotient filter is stored as an array $A[0 \dots m-1]$. Inserting an element with fingerprint f is done by storing one *entry* which is composed of the remainder f_r and three additional *status bits* in the array similarly to the *quotienting* technique used in [9]. Note that since the array is only indexed through the q-bit quotient f_q the number of slots needed for all possible quotients is exactly $m = 2^q$.

During the insertion of an element with fingerprint f there is a possibility of a collision with an already contained element with fingerprint f'. A *hard collision* occurs if the fingerprints are identical $f = f'$. In this case the new fingerprint is not inserted since the fingerprint of the already contained element is indistinguishable from the new one.

If the fingerprints have an equal quotient $f_q = f'_q$ but a different remainder $f_r \neq f'_r$ it is called a *soft collision*. A technique similar to linear probing is used to resolve soft collisions. All entries which belong to fingerprints with the same quotient are stored in a contiguous range in the array which is called a *run*. Different runs always occur in a sorted order in respect to their respective quotients. The entries within a run are sorted by their remainders. During a soft collision the new entry is inserted into its run such that the sorted order is preserved. All following entries from this position until the next empty array slot are shifted by one slot regardless to which run they belong. This means that an entry with quotient f_q is always inserted into a slot $A[f'_q]$ with $f'_q \geq f_q$ and that there are no empty slots between $A[f_q]$ and $A[f'_q]$. An entry is said to be *canonical* or in its *canonical slot* if it was not shifted or equivalently if the remainder f_r of the entry is located in slot $A[f_q]$ where f_q is the corresponding quotient of the entry. A *cluster* is a maximal sequence of consecutive runs such that the only canonical entry is at the beginning of the cluster and it contains no empty slots in between the runs. A *super-cluster* is a maximal sequence of consecutive clusters such that there are no empty slots between them. This means that if the range $A[x], \dots, A[y]$ forms a super-cluster then $A[x]$ is a canonical slot, $x = 0$ or $A[x-1]$ is empty and $y = m-1$ or $A[y+1]$ is empty. The canonical run of an entry is the run in which the entry is located in the quotient filter. Similarly, canonical clusters and canonical super clusters can be defined.

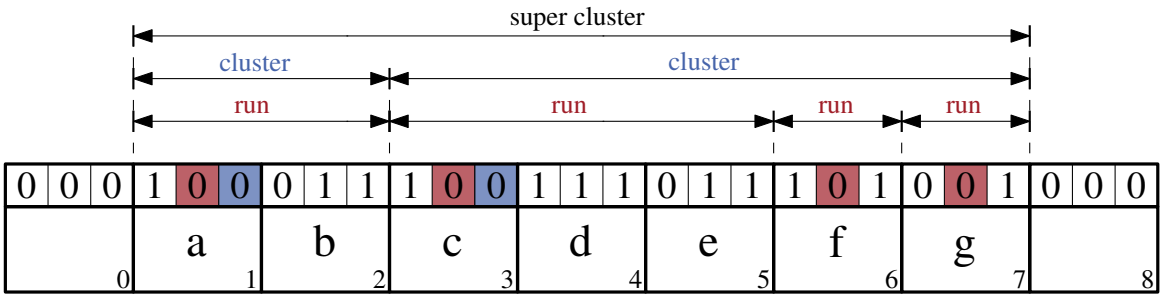


Figure 3.2: Illustration of a run, cluster and super cluster. The bits indicating a run start or cluster start are colored correspondingly.

is occupied	is continuation	is shifted	entry type
0	0	0	empty
1	0	0	cluster and run start (canonical slot)
$*$	0	1	run start
$*$	1	1	continuation of a run
$*$	1	0	-

Table 3.1: Relation of status bits of an entry to its type.

Three additional status bits are use to find the run for a specific quotient even after the entries are shifted during the resolution of a soft collision as can be seen in table 3.1. The first bit is the *is-occupied* bit and is associated with the slot of the entry rather than with the entry itself. This means when shifting an entry this status bit remains unchanged for all involved entries. It is set for the entry in $A[f_q]$ if a fingerprint with quotient f_q is inserted into the quotient filter regardless of which slot it is actually stored in. This allows for an easy and efficient way of checking whether an entry or a run with an associated quotient f_q exists by just checking the is-occupied bit of $A[f_q]$. The second bit is the *is-continuation* bit and is set for any entry which is not at the beginning of its corresponding run. It signals that this entry is continuing the run of the previous entry and is used to determine the start and end of the runs within a cluster. The last status bit is the *is-shifted* bit and is set for all entries that are not in their canonical slots. It is used to find the beginning and end of a cluster within a super-cluster. An illustration of how the status bits relate to runs, clusters and super clusters can be seen in figure 3.2.

The False Positive Rate ε of a quotient filter is the probability that the contains operation reports that an element is present even though it has not been previously inserted. This occurs precisely when there is a hard collision between the fingerprint of the tested element and a fingerprint already present in the QF. Given that the hash function used to generate the fingerprints outputs uniform and independent bits, the probability of a hard collision is

$$\varepsilon = 1 - \left(1 - \frac{1}{2^p}\right)^n \approx 1 - e^{-n/2^p} \leq \frac{n}{2^p} = \frac{n}{2^q} \cdot 2^{-r} = \alpha \cdot 2^{-r} \tag{3.1.1}$$

where n is the number of entries present in the filter and $\alpha = \frac{n}{m}$ is the fill degree. A full quotient filter ($\alpha = 1$) has a maximal false positive rate of $\varepsilon \leq 2^{-r}$ which is illustrated in figure 3.3. Both α and the number of remainder bits r affect the false positive rate.

Note that we restrict all presented quotient filters to entries with at least one remainder bit. Theoretically it is possible to have entries with just three status bits and zero remainder bits. The false positive rate would be equal to the fill degree α, which grows to a 100% as the quotient filter gets filled up completely. We removed this possibility because every run would have a length of at most one. Therefore, there would be no linear probing and one could simply use a bit-vector.

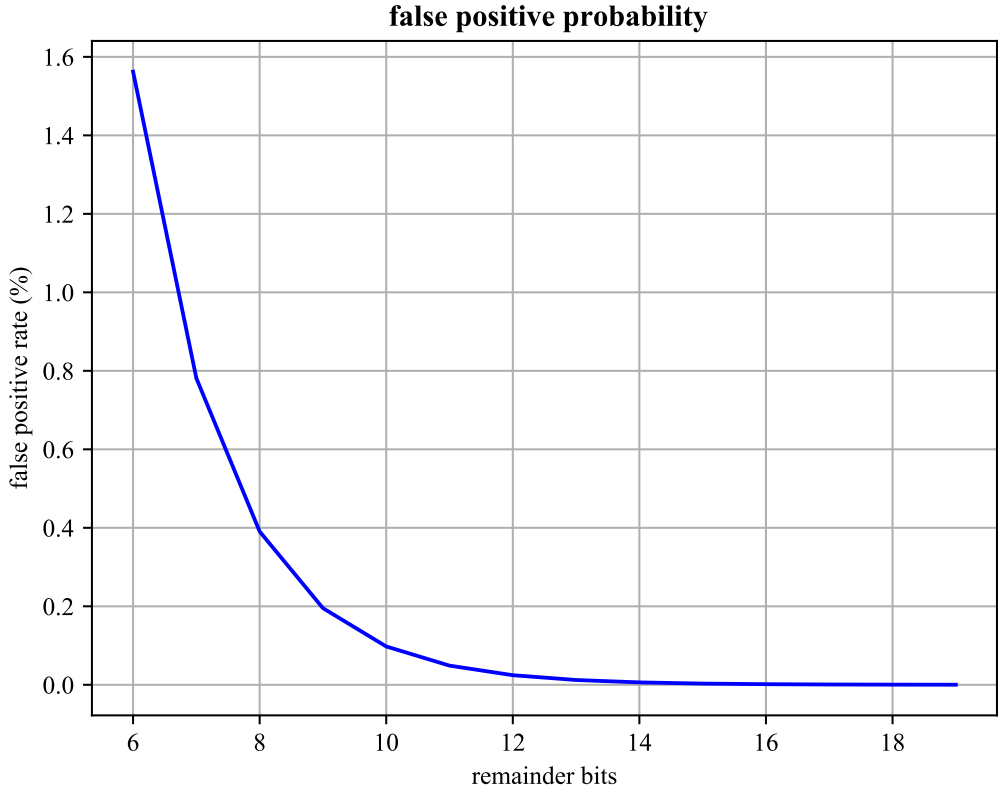
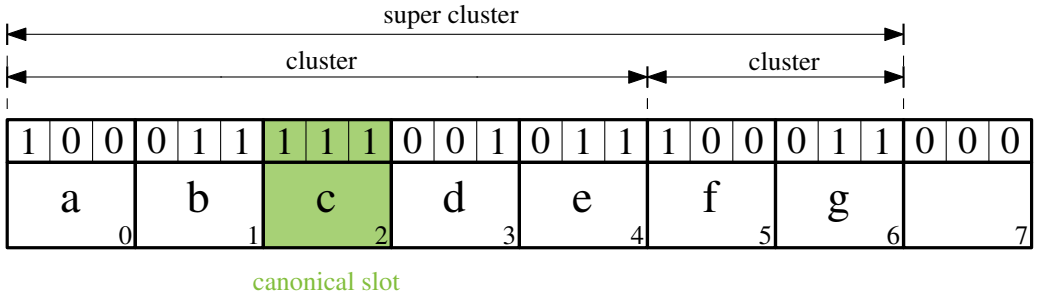


Figure 3.3: The false positive probability $\varepsilon = 2^{-r}$ for a full quotient filter ($\alpha = 1$) with remainder bits in the range of $[6, 20]$.

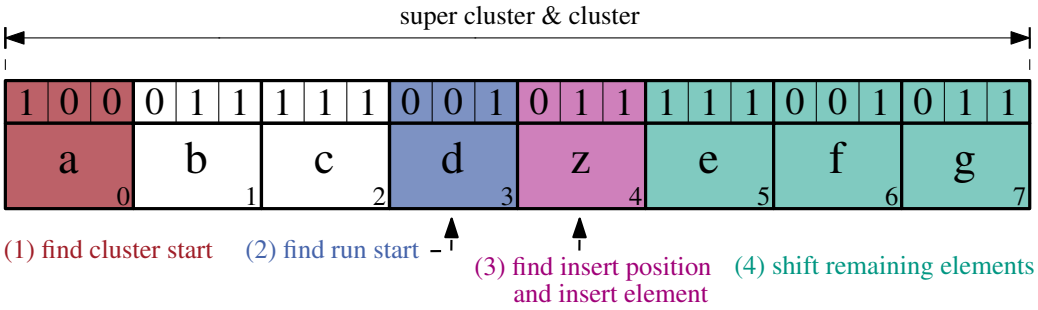
3.1.3 Operations

Insert, Contains and Remove When performing an insert or contains operation on an element with fingerprint f we first need to find the start of the canonical cluster. We scan the array from $A[f_q]$ to the left which means we look at each entry starting at $A[f_q]$ and move towards the beginning of the array one entry at a time. We scan the entries until the first canonical slot is found which is the start of the cluster and also the start of the first run in that cluster. Each run corresponds to one quotient and each element that is inserted marks its quotient through the is-occupied bit. To find the start of the canonical run we traverse the run starts and the occupied-marked slots in a cluster in unison. The run start corresponding to the quotient f_q is found when the occupied-marked slot $A[f_q]$ is reached.

When performing a contains operation we first check if the is-occupied bit of $A[f_q]$ is set. If it is not set then no element with quotient f_q was previously inserted into the quotient filter and the operation can directly return that the element is not present without accessing other entries. Otherwise, the canonical run is located and scanned for a remainder matching that of the given element.



(a) Quotient filter before insertion of an entry with $f_q = 2$ and $f_r = z$ where the order of remainders is $a < \cdots < d < z < e < \cdots < g$.



(b) We first scan for the cluster start from which we then find the start of the canonical run as well as the exact insert position within the run. The new element is inserted and the following entries are shifted to the right.

Figure 3.4: Illustration of inserting an entry into a quotient filter.

The insert operation scans the canonical run for the exact position to insert the new element – to preserve the sorted order in the run – and shifts all following remaining elements as mentioned before. An illustration of an insert operation can be seen in figure 3.4.

The average running time of both operations is bound by the maximal expected super cluster length. As shown in [3] the probability of a super cluster with length greater than k (see below) tends to zero as the size of the quotient filter tends to infinity:

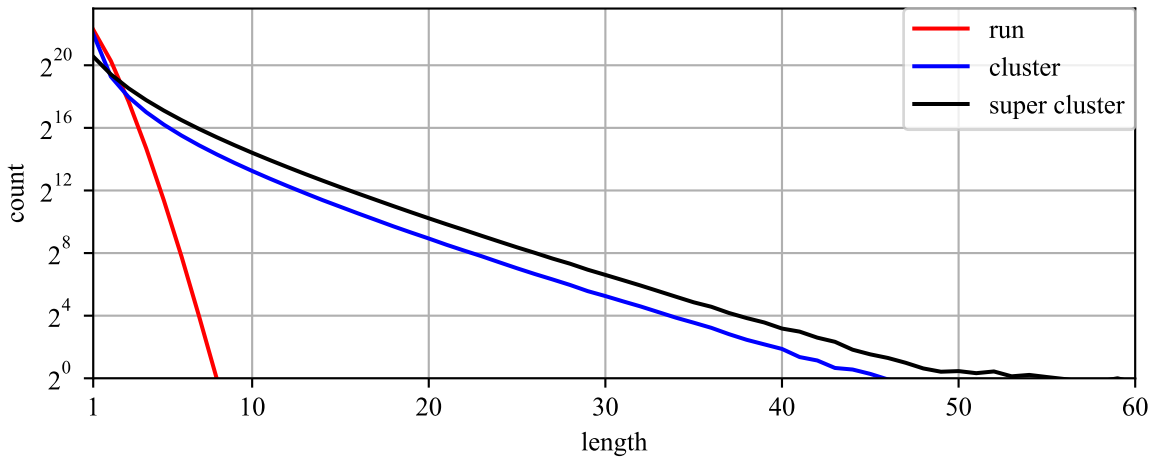
$$\text{For} \quad k = (1+\delta)\frac{\ln m}{\alpha - \ln\alpha - 1}$$

$$\Pr[\textit{there exists a super cluster of length} \geq k] < m^{-\delta} \xrightarrow{m\to\infty} 0$$

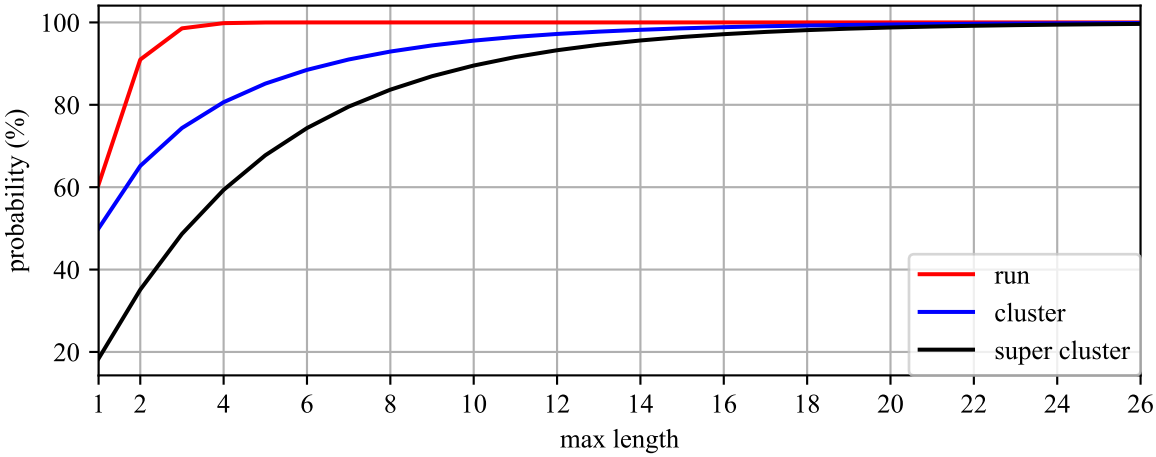
Assuming a fixed maximum fill degree α the average time for contains and insert operations is $O(1)$. Figure 3.5 shows measurements of the number of occurrences of runs, clusters and super clusters of certain lengths. The figure also shows that it is very unlikely to encounter a canonical super cluster of maximal size as the measured maximum length is well below the theoretical limit shown above. In fact about 20% of super clusters are of size 1 and 90% of super clusters have sizes smaller than 10.

A quotient filter also supports the removal of entries which is done similarly to an insert operation with the difference that the entry with matching fingerprint is removed and the

following entries in the super cluster are shifted backwards. Note that in the case of a hard collision where two different elements have the same fingerprint one entry possibly corresponds to multiple inserted elements. Removing an element and its corresponding entry therefore would also remove other inserted elements from the quotient filter as well. To avoid this behavior we can insert duplicate entries into the quotient filter whenever a hard collision occurs. When removing an element only one entry with the matching fingerprint is removed.



(a) Count of runs, clusters and super clusters of a certain length.



(b) Probability of runs, clusters and super clusters of a certain maximal length.

Figure 3.5: Measurements of lengths of runs, clusters and super clusters for a quotient filter with 24 quotient bits, 14 remainder bits and a fill degree of $\alpha = 0.5$. The figures show averages of 100 different measurements. We see that the measured maximum super cluster length is below the theoretical maximum average super cluster length of $\frac{\ln m}{\alpha - \ln \alpha - 1} \approx 86$.

Growing and Merging The ability to dynamically grow allows a quotient filter to be memory efficient not only when all desired elements have already been inserted but also throughout the insertion process. This means that at any point the memory needed to store the QF is linear in the number of the currently inserted elements. Additionally this makes a quotient filter a useful option even in a setting where the exact amount of total elements is not known beforehand or there is no clear upper bound because new elements are added continuously.
Growing a quotient filter works by creating a new array A' which is twice the size of the original quotient filter array A. We then iterate over the whole array and for every non-empty entry e with the remainder f_r and the quotient f_q we construct a new entry e' which we insert into the new array. The quotient and remainder of the new entry are

$$f'_q = 2 \cdot f_q + \left\lfloor \frac{f_r}{2^{r-1}} \right\rfloor \quad \text{and} \quad f'_r = \left\lfloor \frac{f_r}{2} \right\rfloor$$

with r being the number of remainder bits. In effect the most significant bit (*MSB*) of the remainder f_r is moved into the quotient which makes it possible to address twice as many indices in the new array but it also means that the remainders of the new array are one bit shorter. The relative order of entries is preserved and it is possible to copy over the entries in linear time with one pass over both arrays.
Since it is possible that two entries which are in the same run in the original array have different remainders with a different MSBs the quotients of the new entries can be different. They might not be in the same run and possibly not even in the same cluster. Transferring entries from the smaller array to the bigger array in this fashion can thus reduce the distance between their canonical slot and the actual position in the array where they are stored. It is also possible to introduce empty slots in-between entries which are originally in the same super cluster but belong to different super clusters in the grown array. Super clusters can be broken up into multiple super clusters but it can never happen that two super clusters will be merged together. Therefore, transferring one super cluster to the new array will not interfere with the transfer of any other super cluster. Since we skip empty entries growing a quotient filter can be implemented by copying one super cluster at a time. When transferring entries we have to adjust the status bits to reflect the actual run and cluster structure of the new table. Every time a quotient filter grows one remainder bit is converted into a quotient bit. This means a quotient filter with r remainder bits can at most grow r times. To compare quotient filters with dynamic growing to non-growing quotient filters we explicitly refer to them as bounded growing quotient filters (*BGQF*).

Similar to growing, merging two quotient filters can be done in linear time because of the sorted order of runs and entries.

4 Concurrent Quotient Filters

In this chapter we explore different methods to allow concurrent read and write access to a quotient filter. The goal is to create a scalable concurrent quotient filter with minimal memory and performance overhead. To do this in a space efficient manner we first take a look at arbitrary length data types of up to one machine word in length and how to use them in a concurrent setting.

4.1 Compact arbitrary length data types

In the common sequential implementation of the quotient filter described in 3.1 each entry is stored in a separate slot of the array as an integer. Each entry stores three status bits and r remainder bits where r is determined through the intended false positive rate and also by any resizing operations on the quotient filter. This means that the number of bits in an entry will in general not be a power of two and therefore the size of an entry might not exactly coincide with the size of the integer which is used to store it. In this case each integer will have unused bits.

Compact arbitrary length data types are useful to reduce the amount of unused memory. This is done by using the bits of a single integer to hold multiple entries as illustrated in 4.1. Specifically, one integer with I bits can hold $k = \lfloor I/(r+3) \rfloor$ entries with r-bit remainders. In a quotient filter using an array of integers A with k entries per integer the slot $A[x]$ holds the entries corresponding to slots $A[x \cdot k]$, ..., $A[(x+1) \cdot k - 1]$ in a normal quotient filter. If needed, we differentiate between a compact and non-compact quotient filter although every non-compact quotient filter variant discussed in this thesis can be transformed into its compact counterpart without changing any of the theoretical analysis except for the reduced memory footprint.

Note that there still may be unused bits in an integer if the size of an entry does not divide the size of the integer exactly but there are strictly fewer unused bits compared to only

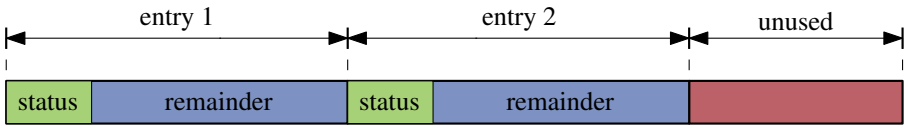


Figure 4.1: Illustration of a single integer holding multiple entries each with status bits and remainder bits. The integer may also contain unused bits.

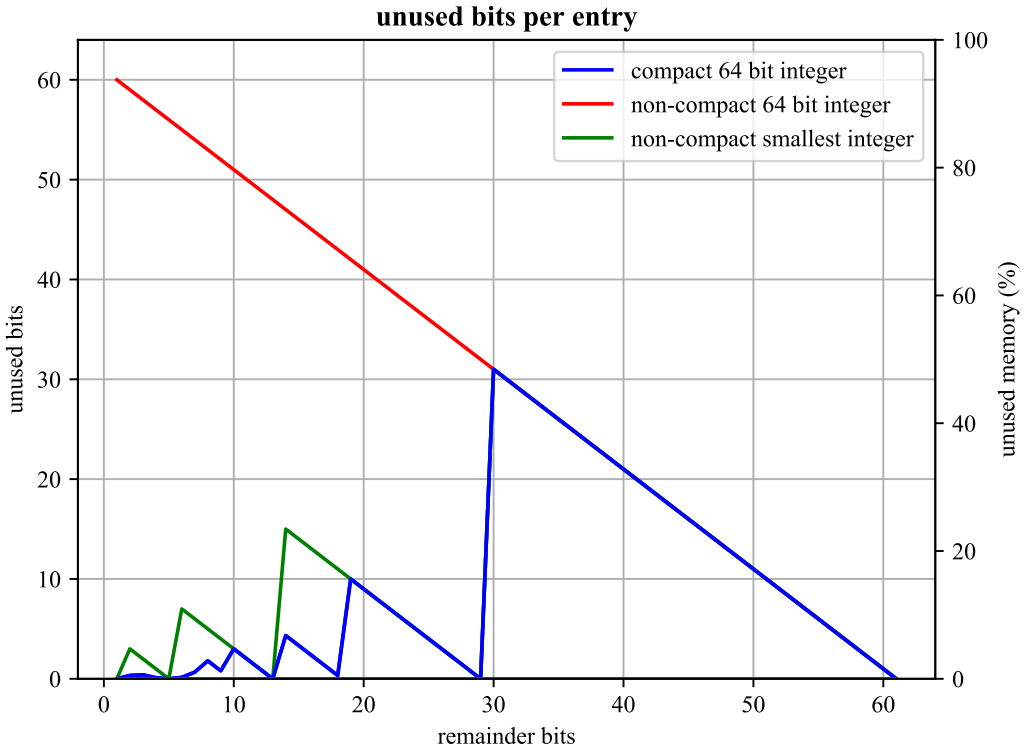


Figure 4.2: The number of unused bits per entry for remainder bits in the range of $[1,61]$ for different storage techniques. The non-compact smallest integer variant always uses the smallest possible integer that has enough space for all remainder bits and three additional status bits.

storing one entry per integer. This is true even if we would always choose the smallest possible integer to store one entry instead of a fixed integer size. For a small number of remainder bits the compact approach is a lot more space efficient as can be seen in figure 4.2.

One possible way to completely avoid unused bits is to allocate a chunk of contiguous memory with a size of exactly $m \cdot (r+3)$ bits for storing m entries with r-bit remainders. This can work for sequential quotient filters but it becomes a problem when this representation of entries is used for concurrent quotient filters as entries may cross the boundaries of a cache line. In this case unaligned compare and swap operations are needed. Additionally, the way that the bits are physically stored in main memory (the endianess of the system) becomes important which can lead to a significant performance overhead while possibly restricting the area of application to only a subset of all available systems. In contrast, when using the compact representation as described above it is possible to allow atomic read and write operations for a whole entry with no functional overhead by simply using an atomic integer instead of a normal integer as the underlying data type.

4.2 Simple Concurrency

In this section we look at simple locking quotient filters and linear probing quotient filters which are simple concurrent quotient filter variants. We also discuss their benefits and shortcomings.

4.2.1 Simple Locking Quotient Filters

The simplest approach to building a data structure that can be used by multiple threads without race conditions or deadlocks is to lock the whole data structure before every access. Obviously, this approach does not scale at all with concurrent accesses by an increasing number of threads. A more scalable variant uses multiple locks, each of which locks a specific non-overlapping range of the quotient filter's array. This can be done in two different ways. Either the number of locks is constant and the number of integers protected by one lock varies with the size of the filter or the number of locks varies and the number of integers in a range remains constant. While Bender et al. explore the former variant [13], we use the latter with a constant number of locks since we want to reduce the memory overhead. In a quotient filter of size m with l locks, lock L_i locks the range $[S \cdot i, S \cdot (i+1) - 1]$ where $i \in 0, \ldots, l-1$ and $S = m/l$ is the range of a single lock. We call this quotient filter variant a simple locking quotient filter (*SLQF*).

When inserting an element with quotient f_q we need to acquire the lock L_i such that f_q is part of the associated range of L_i therefore $i = \lfloor f_q/S \rfloor$. Since an insert operation scans to the left and to the right form position f_q as well as shifts entries in the array, we need to ensure that accessing elements around slot f_q does not result in race conditions or deadlocks.

If we assume that super clusters are smaller than $S/2$ then it suffices to acquire the locks L_i and L_{i+1} in order with $i = \lfloor f_q/S \rfloor$ before starting an insert or contains operation. Accesses to the right of f_q are guarded directly by the locked ranges of the acquired locks. Accesses to the left are only guarded directly if f_q lies in the upper half of the range associated with L_i. Otherwise it is possible that the start of the canonical super cluster of f_q lies in the range of lock L_{i-1}. Note that since the number of slots which we access into any direction is bounded by the size of the super cluster which is smaller than $S/2$ this means that we only access slots which are either in the upper half of the range of L_{i-1}, in the range of L_i or in the lower half of the range of L_{i+1}. Thus if an operation acquires locks L_k and L_{k+1} while another operations acquires locks L_{k+2} and L_{k+3} both can run concurrently without interfering with each other. If the locks of one operation overlap with locks of another than then one of the operations waits for the lock to be released.

Simple locking quotient filters need extra memory to store the additional locks. In addition, they lock large ranges of the array even if only a few contiguous accesses are needed. This prevents them from scaling well beyond a certain number of concurrent accesses. Since the locks are placed on different cache lines to prevent false sharing each lock access incurs a load from main memory which slows down performance.

4.2.2 Linear Probing Quotient Filters

In order to avoid the use of locks altogether and at the same time reduce the memory per entry we introduce a new quotient filter variant called the linear probing quotient filter (*LPQF*). During an insert operation the fingerprint of an element is split into the quotient f_q and the remainder f_r as usual. We then use linear probing with the remainder starting at slot f_q in the array and search for the next free slot to write the remainder into. A contains operation searches for the remainder starting at slot f_q until either f_r is found or an empty slot is found. Note that inserting an element with a fingerprint of zero will not actually insert a new entry since a fingerprint of zero is indistinguishable from an empty slot.

In contrast to standard quotient filters, we only store the remainder bits without any status bits. This approach uses less memory per stored entry which can instead be used to store more remainder bits. It also means that a linear probing quotient filter can easily be converted into a concurrent variant by using an array of atomic integers without the need for additional locks. We can perform atomic loads when scanning the array and atomic compare and swap operations for inserting a remainder.

Eliminating the status bits also leads to the loss of the internal structure of a quotient filter which makes it possible to unambiguously reconstruct the original fingerprint of each entry in the array. There is no longer a distinction between runs, clusters and super clusters. Therefore, when performing a contains operation, all remainders starting from slot f_q until the next free slot are searched for f_r which leads to a false positive rate that is greater than that of a normal quotient filter. Even if we use the 3 saved status bits to store 3 additional remainder bits it may still not be enough to overcome this problem depending on the fill degree. A full LPQF has a false positive rate of approximately

$$\varepsilon = 1 - \left(1 - \frac{1}{2^r}\right)^m \approx 1 - e^{-m/2^r} \leq \frac{m}{2^r} = 2^{q-r}$$

whereas a full normal quotient filter has a false positive rate of $\varepsilon = 2^{-r}$ which is 2^q times lower.

4.3 Advanced Concurrency

In this section we try to design a data structure which forgoes the drawbacks of the previously discussed concurrent quotient filters. We use the structure of quotient filters to our advantage in order to build the advanced concurrent quotient filter (*ACQF*) which only locks local ranges (clusters and super clusters) of the array with no memory overhead.

is occupied	is continuation	is shifted	entry type
0	0	0	empty
1	0	0	cluster and run start (canonical slot)
$*$	0	1	run start
$*$	1	1	continuation of a run
0	**1**	**0**	**write lock**
1	**1**	**0**	**read lock**

Table 4.1: Relation of status bits of an entry to its type for a ACQF with an additional read lock and write lock status.

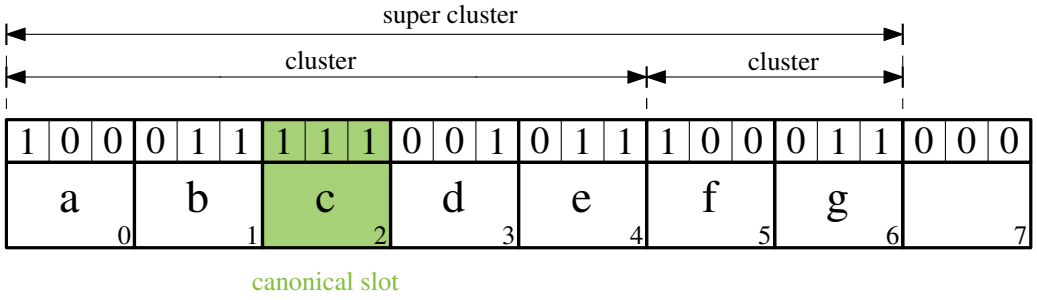
4.3.1 Locking Mechanism

A contains operation only accesses entries which are inside the canonical cluster. Similarly, an insert of an element only accesses entries between the start of the cluster and the end of the super cluster corresponding to the inserted element. If the range of possibly accessed elements of a set of operations are pairwise disjoint then these operations can be performed concurrently. The exclusive access to such a range for one operation can be achieved by locking this range such that no other concurrently running operation may get access to a locked range. Note that if we only perform contains operations then exclusive access is not needed and we can perform the read operations fully concurrently without locking.

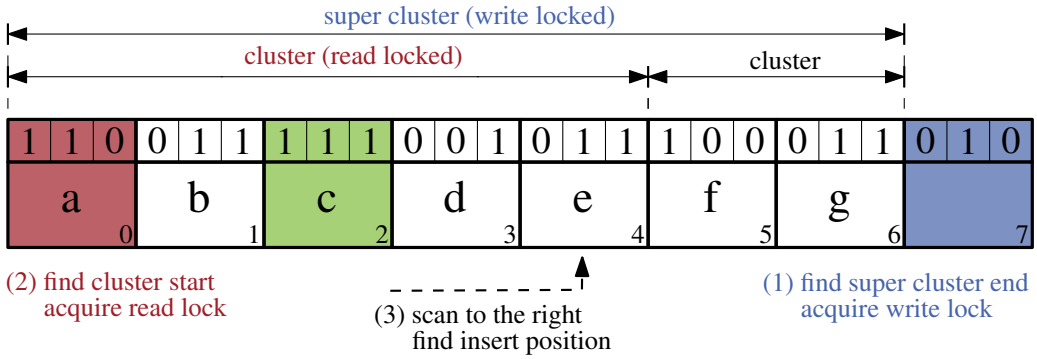
In order to implement locking without any additional memory overhead the status bits in the entries are used. The three status bits allow for a total of eight different combinations (4.1) two of which are unused in the case of a sequential quotient filter. These two remaining combinations will be used to signal the start and the end of a locked range respectively. More precisely the two new states called *read lock* and *write lock* are used to provide exclusive read and write access to a specific subrange in a super cluster. A read lock at $A[r]$ is used to provide exclusive read accesses on the range $A[r,\ldots,x-1]$ where $A[x]$ is the start of the next cluster within the super cluster or $A[x-1]$ is the end of the super cluster. A write lock at the empty slot $A[w]$ immediately after the end of the super cluster gives exclusive write access to that super cluster. If a lock needs to be acquired but the range is already read or write locked then busy waiting is performed until the range is unlocked again. Note that these special locking states overwrite the status of a normal entry which may have to be restored after the insert or contains operation for which the lock was acquired has completed.

4.3.2 Concurrent Operations

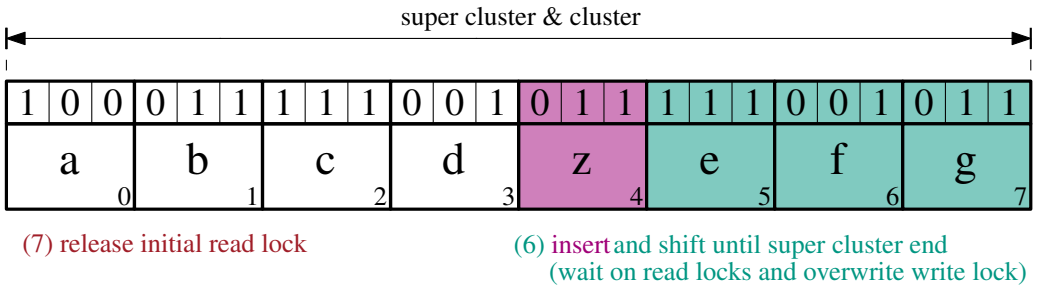
Contains A contains operation only reads the array of the quotient filter and therefore only needs to acquire the read lock. This is done as the first step in the contains operation. First we find the start of the canonical cluster by scanning the array from the canonical slot $A[f_q]$ of the entry to the left until the start $A[r]$ of the current cluster $A[r,\ldots,r']$ is found.



(a) ACQF before insertion of an entry with $f_q = 2$ and $f_r = z$ where the order of remainders is $a < \cdots < d < z < e < \cdots < g$.



(b) First part of concurrent insertion of the entry where the write lock and the initial read lock are acquired. The insert position for the new element in its canonical run are determined.



(c) Second part of concurrent insertion of the entry where the new element is inserted and the following entries in the super cluster are shifted to their final position. The write lock is directly overwritten with the new statuses of the shifted entries which releases the lock implicitly. Note that we have to wait on any cluster starts which are read locked while shifting. In the end the initial read lock is released which completes the insertion.

Figure 4.3: Illustration of an insertion of an entry into an ACQF. Every lock shown is acquired and held by the thread performing the insertion.

The status of $A[r]$ is atomically exchanged to the read lock status with a compare and swap operations. Next the array is scanned to the right until the start of the canonical run is found. The canonical run is scanned and either the searched entry is found or the end of the run is reached if the entry is not present in the quotient filter. This completes the contains operation and the read lock is released restoring the status bits.

Insert An insert operation for an element needs to both read and write to its canonical super cluster, therefore both a read and a write lock need to be acquired. Starting from $A[f_q]$ we first scan right until the first free cell where the write lock is acquired. Afterwards the read lock is obtained and the canonical run is found just as in the contains operation. In the canonical run the correct insert position for the new entry is located. The general idea for inserting the new entry and shifting the following entries in the super cluster is the same as in the sequential variant since the range of elements we modify is fully protected by our locks. We write the current entry to its final position with a possibly modified status by swapping it with the entry at that position. Then we increment the position and continue swapping entries into their final position until we reach the end of the super cluster. Note that all swaps still have to be done through atomic operations.
The write lock gives exclusive write access to the whole super cluster and remains the same until it is overwritten by the last swap at the end of the super cluster at which point it is considered to be released. The read locks on the other hand are only valid for one cluster. If the canonical cluster is not the last cluster of its super cluster then we will reach the start of a new cluster during the shifting of the entries. If the cluster is read locked we wait otherwise we just overwrite the entry with its new status and the shifted entry. The initially acquired read lock is held until all entries have been shifted.
When the end of the super cluster is reached, the dummy entry which holds the write lock will be overwritten by the last shifted entry, at which point the write lock is automatically released. The insert operation is therefore finished and the read lock can also be released. An illustration of the whole concurrent insertion operation can be found in figure 4.3.

Lock free operations When a contains operation is performed and the slot $A[f_q]$ is empty this means that the element is not present in the quotient filter and there is no further need to access other entries or acquire a read lock. Similarly, if $A[f_q]$ is empty when performing an insert then we just perform an atomic compare and swap operation without needing to acquire any locks. If the quotient filter uses the compact representation of entries then we can further improve the insert operation even if $A[f_q]$ is not empty. In the case where the next empty entry is stored in the same integer as f_q we can also use a simple compare and swap operation to insert the new element in the super cluster without the need for any read or write locks. If the compare and swap operations for these optimizations fail we fall back to the approach that uses read and write locks as described above.

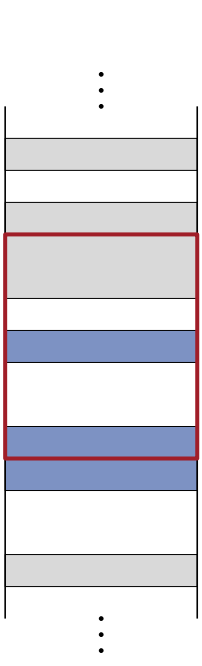
4.4 Concurrent growing

4.4.1 Overview

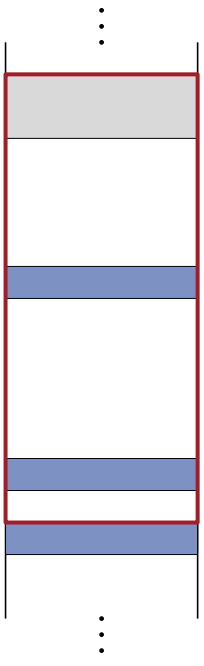
When growing an ACQF the basic procedure is the same as for a sequential quotient filter 3.1.3. In order to take advantage of multiple threads being active during a grow operation copying the entries is done in a blockwise fashion as can be seen in figure 4.4. The array is divided into blocks of size B and each thread participating in growing the quotient filter handles one block at a time until no blocks are left and all entries have been transferred over to the new array. As mentioned before, we move each super cluster separately. To ensure that each super cluster is only copied once we assign the super clusters to the block in which they start and a thread handles all super clusters assigned to its block completely even if they surpass the block boundaries into other following blocks. Since each super cluster has exactly one distinct starting entry there are no ambiguities as to which thread handles which super cluster.
One problem that can occur, however, is that while some threads are already starting to grow the quotient filter, other threads mights still perform contains or insert operations. While growing, read operations can be done simultaneously, as long as read locks are acquired and released from the growing threads. Performing an insert operation into a super cluster that is currently being copied or has already been copied can lead to the loss of this insert. A way to overcome this problem is to use the fact that we copy the entries one super cluster at a time and that write locks also grant exclusive write access for a whole super cluster. Therefore, acquiring a write lock prior to transferring a super cluster prevents simultaneous insertions into that super cluster. This write lock is never released in order to also prevent insertions after the super cluster is already copied. While this solves the problem of unwanted insertions this also leads to deadlocks on inserts because insert operations perform busy waiting when encountering a write lock and since the write lock is never released a deadlock occurs. To distinguish between a write lock used for inserts and a write lock used for growing we use the fact that write locks are always applied to empty entries – entries where the remainder is zero – and that we always have at least one remainder bit in each entry. This allows us to differentiate the write locks based on the remainder of the entries. We have the insert write lock with a remainder of zero and a new growing write lock with the remainder set to one.

4.4.2 Implementation

Since it is possible that one thread still executes an insert or contains operation on an old quotient filter which has already been grown into a bigger QF, where new operations are already performed, there is a need to keep the old quotient filters in memory until we can guarantee that no thread is accessing them anymore and no further operations are performed on them. To accomplish this we use a wrapper data structure around a normal quotient filter which manages growing operations and also allocates memory for the new array while deallocating old arrays when they are no longer in use. In the concurrent case this is can be done with hazard pointers or other memory reclamation techniques. The wrapper data structure makes sure that only one thread can initiate the growing process at a time. It also ensures that other threads accessing the quotient filter while growing will be redirected to take part in the growing process before they start their insertion or contains operation.



(a) View of the block which is copied and the super clusters associated with it as well as super clusters associated with other blocks.



(b) View of the copied block and its super clusters in the grown quotient filter.

Figure 4.4: Illustration of the blockwise concurrent growing operation.

4.5 Compact Concurrent Quotient Filters

Using compact data types as discussed in 4.1 reduces the amount of memory needed to store the same amount of entries, since multiple entries are stored in one integer as opposed to just storing one entry in each integer of the array. This is true regardless of whether the data types are atomic or not.
The slight increase in complexity and computational effort needed to read and write a single entry is compensated by the fact that reading and writing multiple consecutive entries in the array is more efficient. While reading one entry in an integer all other entries are also loaded from memory. We perform all computations on the already loaded entries and then write back all entries in the integer with one memory access. In effect, computations are performed on each integer separately with each integer requiring only one read from and possibly one write to main memory. The number of accesses to main memory are reduced by the same factor as the memory usage is reduced. For the same reason prefetching and caching are also more efficient.

5 Fully Dynamic Quotient Filters

The false positive rate of a quotient filter which stores entries with r remainder bits will increase as more elements are inserted into the filter until it is filled completely, then the false positive rate equals 2^{-r}. A bounded growing quotient filter grows by using one of its remainder bits as an additional quotient bit which doubles the number of elements that the BGQF can hold. This also means that each time a full BGQF grows and is filled up again the false positive rate doubles. If it continues to grow until all remainder bits are used as quotient bits the BGQF reaches its maximum size and the false positive probability becomes 100%.

The goal of a fully dynamic quotient filter (*FDQF*) is to both have an upper bound on its false positive rate like a common quotient filter and at the same time be able to dynamically grow like a growing quotient filter. To achieve this it uses multiple growing quotient filters in levels similar to Scalable Bloom Filters [2].

5.1 Overview

5.1.1 Overall Structure

A fully dynamic quotient filter can have multiple levels each of which is a quotient filter. An illustration can be seen in figure 5.1. In the beginning there is only one level L_0 with a maximum false positive rate of ε_0. We insert elements into this level until we hit a predefined maximum fill degree $\alpha_{\max}$ at which point a new level L_1 is added. Further insertions only insert into the last added level. When the quotient filters fill up to their

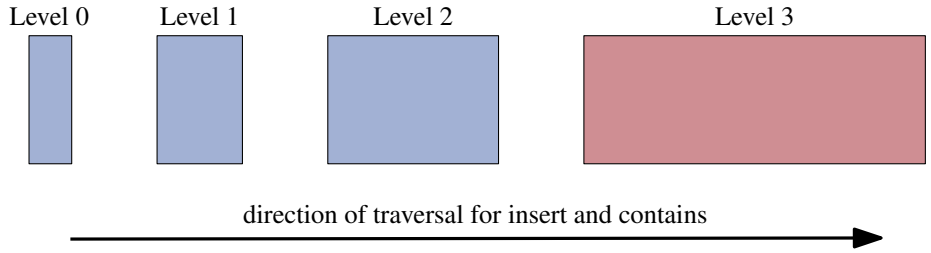


Figure 5.1: Illustration of a fully dynamic quotient filter with multiple levels where the size of each level doubles compared to the last. Regular inserts are only performed into the last level. The previous levels either don't participate in insert operations or only use quick insert.

maximum fill degree we keep adding new levels L_i with a maximum false positive rate ε_i. Adding new levels when needed gives a similar effect as dynamic growing. Since the elements are distributed over all existing levels a contains operation has to check each level starting at L_0 for the membership of the given element.

Given a user defined maximum false positive rate $\varepsilon_{\max}$ we show that we can choose the false positive rates ε_i for each level such that the combined false positive rate ε of the fully dynamic quotient filter is lower than $\varepsilon_{\max}$. Specifically, we choose the maximum false positive rates of the levels as $\varepsilon_0 = \varepsilon_{\max}/2$ and $\varepsilon_{i+1} = \varepsilon_i/2$. Halving the false positive rate of a level compared to the previous level is achieved by simply using an additional remainder bit.

At any given time the false positive rate of every level L_i in a FDQF is bound by its maximum false positive rate ε_i. We can find a bound for the total probability of a false positive for a FDQF as follows: with a probability of at most ε_0 a false positive occurs in L_0 and with a probability of at least $1-\varepsilon_0$ we continue to check the next level for membership. We define the probability of a false positive occurring in level L_i or later as $\bar{\varepsilon}_i = \varepsilon_i + (1-\varepsilon_i)\cdot\bar{\varepsilon}_{i+1} \leq \varepsilon_i + \bar{\varepsilon}_{i+1}$. Therefore we have

$$\varepsilon = \bar{\varepsilon}_0 \leq \varepsilon_0 + \bar{\varepsilon}_1 \leq \varepsilon_0 + \varepsilon_1 + \bar{\varepsilon}_2 \leq \sum_{i=0}^{l} \varepsilon_i$$

with l being the number of existing levels. Since $\varepsilon_i = \varepsilon_0 \cdot 2^{-i}$ and $\varepsilon_0 = \varepsilon_{\max}/2$ we have

$$\varepsilon \leq \sum_{i=0}^{l} \varepsilon_i \; = \varepsilon_0 \cdot \sum_{i=0}^{l} 2^{-i} \; \leq \varepsilon_0 \cdot \sum_{i=0}^{\infty} 2^{-i} = \frac{\varepsilon_{\max}}{2} \cdot 2 = \varepsilon_{\max}$$

which concludes the proof.

5.2 Level Structure

In this section we show how to choose the size of the levels and how to support growing quotient filters as levels.

5.2.1 Level Size

In the previous section we showed how to choose the maximum false positive rates for each level in order to satisfy the given upper bound on the total false positive rate of the fully dynamic quotient filter. To support these false positive rates there is a lower bound to the number of remainder bits used for the quotient filter one each level. The size of a level is completely independent from the number of its remainder bits. To reduce the number of overall levels, we double the size of each new level. This translates to one additional quotient bit per level. This assures that for a given maximum total capacity we only have a logarithmic number of levels.

5.2.2 Growing Levels

Instead of using a common quotient filter for each level we use a bounded growing quotient filter with the additional constraint that it can only grow a fixed number of times. By adding this constraint to the BGQF we have effectively bound its false positive rate. Compared to a common QF this allows us to have more control over the behavior of the FDQF. After the fixed number of grow steps g are completed on any level L_i its size $m_i = 2^{q_i}$ should be twice the size of the proceeding level. Which means the initial size of L_i is $m_i \cdot 2^{-g} = 2^{q_i - g}$ where $q_i - g$ are the number of quotient bits at initialization. Since the maximum false positive rate of a level $\varepsilon_i = 2^{r_i}$ is halved for each new level the remainder bits r_i at the end of growing have to increase by one for each new level. The additional quotient bits needed for growing are obtained by using the remainder bits. Therefore, the initial remainder bits for level L_i are $r_i + g$. Note that after a level has grown g times to its final size it is equivalent to a non-growing quotient filter.

The choice of g effects both running time as well as memory usage. For small g we don't need to grow a single level very often but in turn the initial size of a level might be quite big. This means when a new level is just created there is a lot of unused memory which has to be filled up first through further inserts. Large values for g have the opposite effect. The initial size of a new level is smaller because only a small amount of memory gets added compared to the whole FDQF. But the level has to grow more often making insertions slower on average.

5.3 Quick Insert

We only fill up each level until it reaches a maximum fill degree of $\alpha_{\max}$. This is beneficial because the performance of a quotient filter decreases as the fill degree increases. The downside is that the remaining space is left unused. In order to not waste the empty slots which are still available, when adding a new level, we use a new form of insertion on each level called *quick insert*. A quick insert only inserts an element into the quotient filter if this is possible without shifting any elements. This means that either the fingerprint corresponding to the element is already contained or the slot $A[f_q]$ is empty. A quick insert never increases the size of a cluster as entries are only added if they form a new cluster of length one. This means that adding entries through a quick insert does not degrade the performance of future contains operations.

The general insertion of a new element into a FDQF with l levels works as follows: we perform a potential quick insert for all of the first $l-1$ levels in order. We stop if one of the inserts succeeds. If no quick insert succeeds we perform a regular insert into the last level L_l which may add a new level.

5.3.1 Concurrent Optimization

Note that when level L_i is added any insert into any of the proceeding levels is only done through quick inserts. Since a quick insert only writes an entry into $A[f_q]$ if this slot is empty this can be done through one atomic compare and swap operation without the need for any read or write locks. If it succeeds, the slot was previously empty which also means that the fingerprint was not contained previously. If the compare and swap operation fails then it is still possible that the fingerprint is contained in the quotient filter and we have to perform a contains operation. Because we don't use any locks on any inserts we can also omit the locks on all contains operations on lower levels, both the one that is part of a quick insert as well as any further contains operations. Only the last level needs any locks for concurrent insert and contains operations while the other levels can be accessed without locks.

5.3.2 Contains Optimization

Using quick inserts also allows us to make a change to the contains algorithm. This increases the performance of a contains operation for fully dynamic quotient filters. For this we prove the following theorem:

Theorem 5.3.1. *If an element e is inserted in level L_x then none of the previous levels $L_0, \ldots, L_{x-1}$ contain e or have an empty canonical slot corresponding to e.*

Proof. Let $A_i[f_{q_i}]$ be the canonical slot of the element e in level L_i and assume we insert the element into level L_x. We use the quick insert technique described above. Therefore, we go through every level starting from L_0 and try to perform a quick insert. We stop the insertion if the quick insert succeeds which only occurs if e is already contained or if the canonical slot $A_i[f_{q_i}]$ is empty and we can insert the entry without shifting. This means if we reach L_x then no previous level can contain e or still have an empty canonical slot corresponding to e. □

We can use this theorem to add another condition in which we can stop the contains operation on a FDQF without needing to look at all existing levels. Specifically, we stop the contains operation as soon as we find a level which either contains the given element or has an empty canonical slot corresponding to the element.

5.3.3 Analysis

The time of an insert operation for a FDQF depends mainly on the number of levels we have to access. If we do not use quick inserts we only insert into the last level, therefore, the insertion for a FDQF takes the same time as a single insertion into one level. When using quick inserts we try to insert into all of the levels until a quick insert succeeds or we perform a regular insert into the last level. We also refer to the last level of a FDQF as the active level. In this section we try to determine how many levels we need to access on average.

Let i be the number of total insertions into the FDQF. Because inserting a fingerprint which is already present in one of the levels does not change the data structure, we only count insertions that add a new entry. This can either be the result of inserting a duplicate element or through hard collisions. When level L_k is created, all inserts that reach this level are regular inserts until we reach its maximum fill degree $\alpha_{\max}$. At this point we create level L_{k+1} and all further insertions into L_k are made through quick inserts. Let C_k be the number of insertions that happen before we create level L_k. Note that $C_0 = 0$ since we create the first level before the first insertion. The level L_k is the active level as long as $C_k < i \leq C_{k+1}$.

A regular insert into L_k will always insert the given element into the level. For all previous levels we perform quick inserts which have a chance of failing. Specifically, a quick insert fails if the canonical slot of the given element is not empty. If a quick insert fails we continue the insertion operation at the next level. Thus we can define the probability $I_k(i)$ of inserting the i-th element into level L_k given that we reached L_k as:

$$I_k(i) = \begin{cases} 1 & \text{, if } L_k \text{ is active} \\ 1 - \alpha_k(i) & \text{, otherwise} \end{cases}$$

where $\alpha_k(i)$ is the expected fill degree of L_k before the i-th insert. We also need to define the probability $R_k(i)$ of reaching level L_k on the i-th insert. We reach a level if we already reached the previous level but the insertion failed. Thus it can be defined as:

$$R_k(i) = R_{k-1}(i) \cdot (1 - I_{k-1}(i)) = \prod_{j=0}^{k-1} \left(1 - I_j(i)\right).$$

Since an insert can only fail for level L_k if $C_{k+1} < i$ we get:

$$R_k(i) = \prod_{j=0}^{k-1} \alpha_j(i).$$

Let $E_k(i)$ be the expected number of elements in L_k before the i-th insert. Note that $\alpha_k(i) = E_k(i)/m_k$, therefore, we write:

$$R_k(i) = \frac{1}{M_k} \cdot \prod_{j=0}^{k-1} E_j(i)$$

with $M_k = \prod_{j=0}^{k-1} m_j$. The number of elements in a level grows by one if we reach this level and we successfully insert an element into it where the probability of a successful insertion is given by $I_k(i)$:

$$E_k(i+1) = E_k(i) + R_k(i) \cdot I_k(i)$$

where $E_k(C_k) = 0$ when level L_k is created. By using the definition of $I_k(i)$ we get:

$$E_k(i+1) = \begin{cases} E_k(i) + R_k(i) & \text{, if } L_k \text{ is active} \\ E_k(i) + R_k(i) \cdot \left(1 - \dfrac{E_k(i)}{m_k}\right) & \text{, otherwise} \end{cases}$$

The first case describes $E_k(i+1)$ for regular inserts. While L_k is active, every element that was not successfully inserted into any of the previous levels is inserted into L_k. We can rewrite the first case as $i - S_k(i)$ where $S_k(i) = \sum_{j=0}^{k-1} E_j(i)$ is the sum of expected elements in the levels up to L_k. Given that $E_k(C_{k+1}) = \alpha_{\max} \cdot m_k$ is the point where we switch from regular inserts to quick inserts, we can solve the recurrence in the second case to yield:

$$m_k \left(1 - (1 - \alpha_{\max}) \left(1 - \frac{R_k(i)}{m_k}\right)^{i - C_{k+1} - 1}\right).$$

We can further simplify this case by defining $\beta = 1 - \alpha_{\max}$. In summary we have:

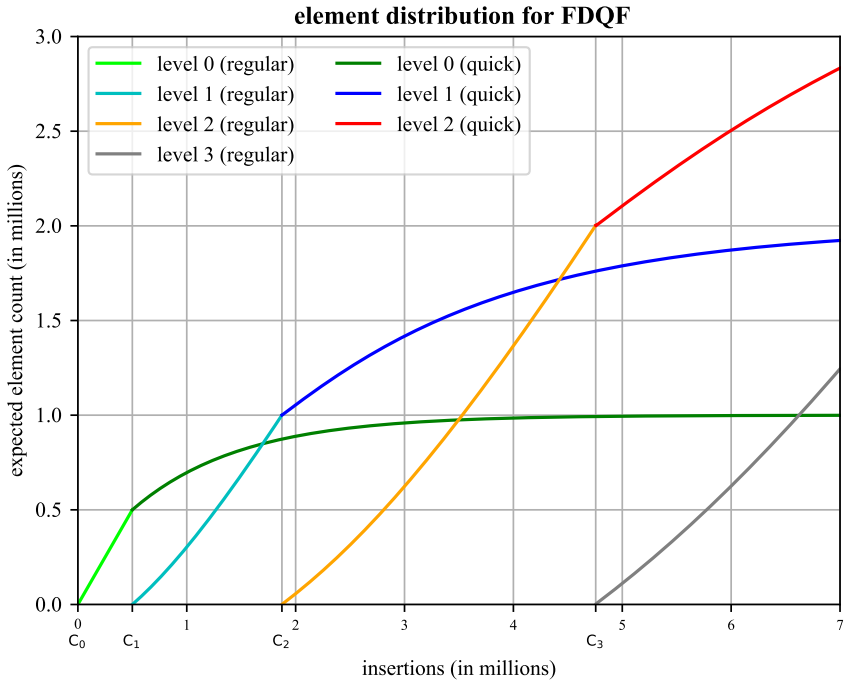
$$E_k(i+1) = \begin{cases} i - S_k(i) & \text{, if } L_k \text{ is active} \\ m_k \left(1 - \beta \left(1 - \dfrac{R_k(i)}{m_k}\right)^{i - C_{k+1} - 1}\right) & \text{, otherwise} \end{cases}$$

The expected distance $D(i)$ of levels we have to traverse to insert an element is:

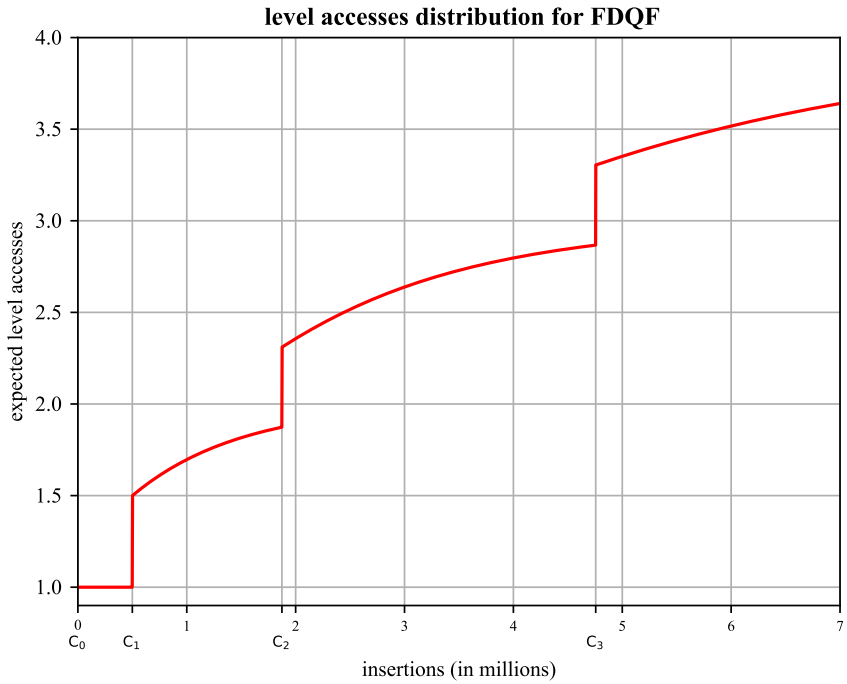
$$D(i) = \sum_{j=0}^{l_{\max}(i)} R_j(i)$$

where $l_{\max}(i)$ is the active level at the time of the i-th insert.

A visualization of how $E_k(i)$ and $D(i)$ behave can be found in figure 5.2. We can see that the levels keep filling up relatively quickly even after they become inactive compared to when they are active. This means that the memory of each level is used more efficiently but also that an insert or contains operation likely looks at almost all levels. If we double the maximum capacity for each level compared to the previous one such that $m_{k+1} = 2 \cdot m_k$ then the number of levels will be logarithmic in the number of total insertions. In this case, an insertion into a FDQF has logarithmic running time in the total number of stored elements.



(a) The expected element counts for each level with insertions split into regular inserts and quick inserts for each level.



(b) The expected number of levels we need to access until an insert operation succeeds.

Figure 5.2: Expected behavior of a fully dynamic quotient filter. The maximum capacity for the first level is $m_0 = 1\,000\,000$ and doubles for each further level. We insert $7\,000\,000$ elements in total and set the maximum fill degree to $\alpha_{\max} = 0.5$. The expected number of inserts C_k which are needed to create the k-th level are: $C_0 = 0$, $C_1 = 500\,000$, $C_2 = 1\,873\,374$, $C_3 = 4\,753\,549$.

6 Experimental Evaluation

In this chapter we are going to experimentally evaluate the different variations of quotient filters described throughout this thesis. We compare them against each other and show their benefits and shortcomings. We first describe the environment and the process used in creating the measurements. Then we present the results for the concurrent quotient filters followed by the growing quotient filters.

6.1 Setup

6.1.1 Environment

All experiments are done on a two socket system equipped with two Intel Xeon E5-2650 v2 CPUs with 8 cores each (16 with hyper threading) running at a clock speed of 2,6 GHz. All quotient filter variants are implemented in C++17 and compiled with gcc 8.2.0 with optimization level -O3.

6.1.2 Measurement

The performance measurements in the following sections are done by measuring the time of performing a given number of operations. The elements which are being inserted are 64 bit integers with values chosen at random from an uniform distribution over all possible integer values. The hash function used for all measurements is xxHash [1]. Because the performance of insert and contains operations may vary based on the actual values of the elements we repeat all measurements nine times. These nine measurements use three different random seeds such that each random seed is used for three of the nine measurements. All results are then averaged to give one final running time value.

For the purpose of the following experiments we split the contains operations into two different kinds: successful and unsuccessful contains operations. Successful contains operations test the quotient filter with elements that have been inserted previously which means that the quotient filter will always return that the given element is present. Unsuccessful contains operations use new random elements. The quotient filter will only report the presence of such an element if the element randomly matches one of the inserted elements or if there is a false positive because of a hard collision on the fingerprint of the element. The

[1] `https://github.com/Cyan4973/xxHash`

probability of the first case occurring is $n \cdot 2^{-64}$ where n is the number of inserted elements. This is negligible for practical values of n. Thus, we use the results of the unsuccessful contains operations to determine the false positive rate of the quotient filter.

When performing concurrent operations we first create an array of all elements which will be inserted. These elements are then processed in blocks of 4096 where each thread processes one block at a time until no blocks are left and all elements have been processed. All concurrent experiments are performed with 16 threads unless stated otherwise.

Every tested instance of linear probing quotient filters has an additional 3 remainder bits compared to the other quotient filters to compensate for the missing status bits such that they have the same amount of memory as the quotient filters they are compared with.

6.2 Concurrency

In the first part of the experimental evaluation we look at the different variations of concurrent quotient filters presented in section 4: simple locking quotient filters (SLQF), linear probing quotient filters (LPQF) and advanced concurrent quotient filters (ACQF). Each variant is both evaluated with and without compact arbitrary length data types. We compare them against each other and measure the impact of their differences on performance. We also evaluate their speedup and efficiency when using multiple threads.

6.2.1 Speedup and Bloom Filter Comparison

In this section we compare the different concurrent quotient filter variants presented in this thesis (SLQF, LPQF, ACQF) and show their speedup in relation to a common sequential quotient filter. We also compare them to a concurrent Bloom filter which uses 7 hash functions and a bit array with 2^{28} bits which corresponds to half of the memory usage of the compact quotient filter variants. First we measure the running time for the insert operations of 14.000.000 elements into an empty filter with a maximum capacity of 2^{25} entries. After the quotient filter has been filled to 40% of its capacity we measure the successful and unsuccessful contains operations separately. The results can be seen in figure 6.1.

Because the LPQF uses no locks and does not need to find the start of the cluster, it is consistently faster than the ACQF and SLQF. While the ACQF locks only small ranges of the array, the SLQF always locks large ranges which increases the probability of contention on the locks. Additionally, the SLQF stores its locks in a separate array which means that every time a lock is acquired or released, an additional cache miss occurs. These differences are clearly visible in our measurements where the LPQF can insert up to 200 million elements per second with 16 threads, whereas the ACQF and LPQF reach only 150 and 50 million insertions per second respectively.

Both types of contains operations are faster than inserts on average since they only need to modify a small number of entries for locking if any at all. Between the two contains operations the unsuccessful operation is faster for the ACQF and SLQF because we don't necessarily need to access multiple entries if there is no canonical run for the given element. Successful operations are guaranteed to have a canonical run since the element has been inserted before. The LPQF does not store status bits and lacks the internal structure of a normal quotient filter. Therefore, this optimization can not happen and we search through all elements until either an empty slot or the given element is found. This is why successful contains operations are faster on average for the LPQF. These opposite behaviors between the two contains operations for the ACQF and LPQF are also reflected in their relative

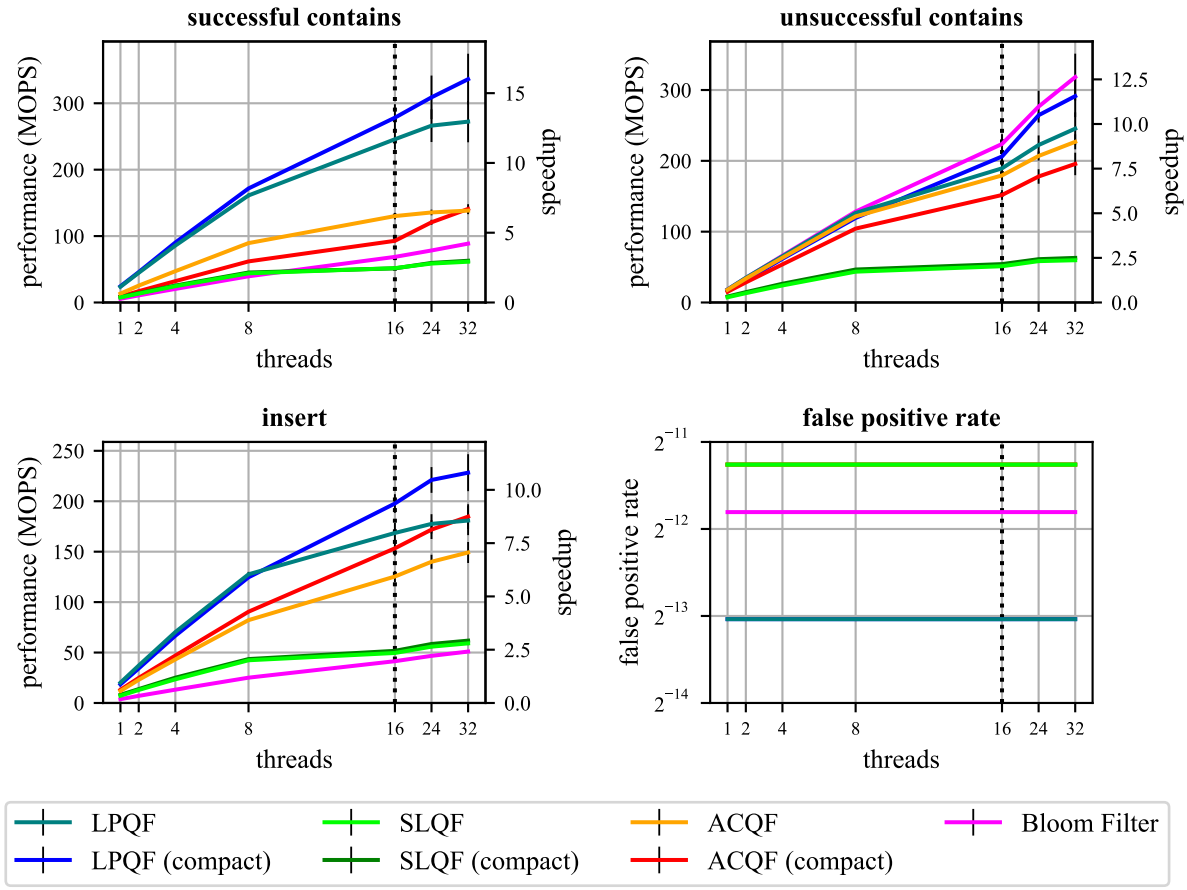


Figure 6.1: Measuring the performance of different concurrent quotient filters and a concurrent Bloom filter with varying number of threads. We also show the speedup relative to a sequential quotient filter. The dashed line indicates that hyper threading is used for measurements with 24 and 32 threads (note the different x-axis scale). In the false positive rate plot the blue line includes both LPQF variants while the green line includes all SLQF and ACQF variants. The used parameters are: $n = 14\,000\,000$, $m_{QF} = 2^{25}$, $\varepsilon = 2^{-10}$, Bloom filter bits $= 2^{28}$, Bloom filter hash functions $= 7$.

performance differences compared to insertions. The LPQF is up to 2.5 times faster than the ACQF on successful contains whereas the unsuccessful contains operations are only 10% faster.

We can also see the differences between compact and non-compact variants of the concurrent quotient filters. The performance of the SLQF variants is unaffected by using a compact representation as the overhead of locking is too great thus overshadowing any differences. The LPQF benefits from the compact representation resulting in increased performance for all operations with a growing number of threads. This is also true for insertions into the ACQF whereas both contains operations have a decreased performance when using the compact variant. The differences in performance using 16 threads are largest for insert operations with a 20% increase in performance for the compact variant for both the ACQF and the LPQF.

Similarly to the LPQF, the Bloom filter does not use any locks. But because for each of its hash functions a different random bit is read or written, each of these accesses incur a cache miss. This slows the Bloom filter down significantly for inserts and successful contains operations where its performance is very similar to the SLQF. Unsuccessful contains operations can abort early as soon as one of the checked bits is not set, which improves the performance. This optimization works very well as the Bloom filter has better performance than the other tested quotient filters for this operation where it is up to 10% faster than the compact LPQF.

The figure also shows the speedup compared to the sequential non-compact quotient filter. The ACQF and LPQF filters with 2 threads are about as fast as the single threaded sequential variant. As more threads are added we can see that all concurrent variants consistently scale linearly up to 8 threads. At this point the successful contains operations of the SLQF, ACQF and LPQF have a speedup of about 2.5, 4 and 8 respectively. For more than 8 threads the second socket on the machine is used because each socket has 8 cores on the machine that runs the experiments. This is the reason for the clear drop of the performance per thread that can be seen at the 8 thread mark. For more than 16 threads hyper threading is used which is why the performance gain per additional thread is again reduced. This is why we settle on using 16 threads for all other experiments in the following sections if not stated otherwise.

The false positive rate does not depend on the number of threads. Since the LPQF has an additional three remainder bits compared to the other two concurrent filters its false positive rate is lower for the fill degree used in this evaluation.

6.2.2 Fill Degree

In this section we evaluate the performance of the different quotient filters with various fill degrees. Figure 6.2 shows the performance of insert and contains operations for fill degrees in the range of 10% to 90% in increments of 10% steps. For each of the tested fill degrees we first filled the QF exactly to the current fill degree. Then the successful and unsuccessful contains operations are measured followed by the insert performance. The performance of each operation is determined by measuring the running time for 100 000 elements.

The performance of the SLQF is very stable for all operations up to a fill degree of about 60%. For successful contains operations the ACQF is also very stable whereas the performance of the LPQF drops by 20%. The performance for the unsuccessful contains operations and insertions of the ACQF and LPQF at 60% fill degree is only half as large

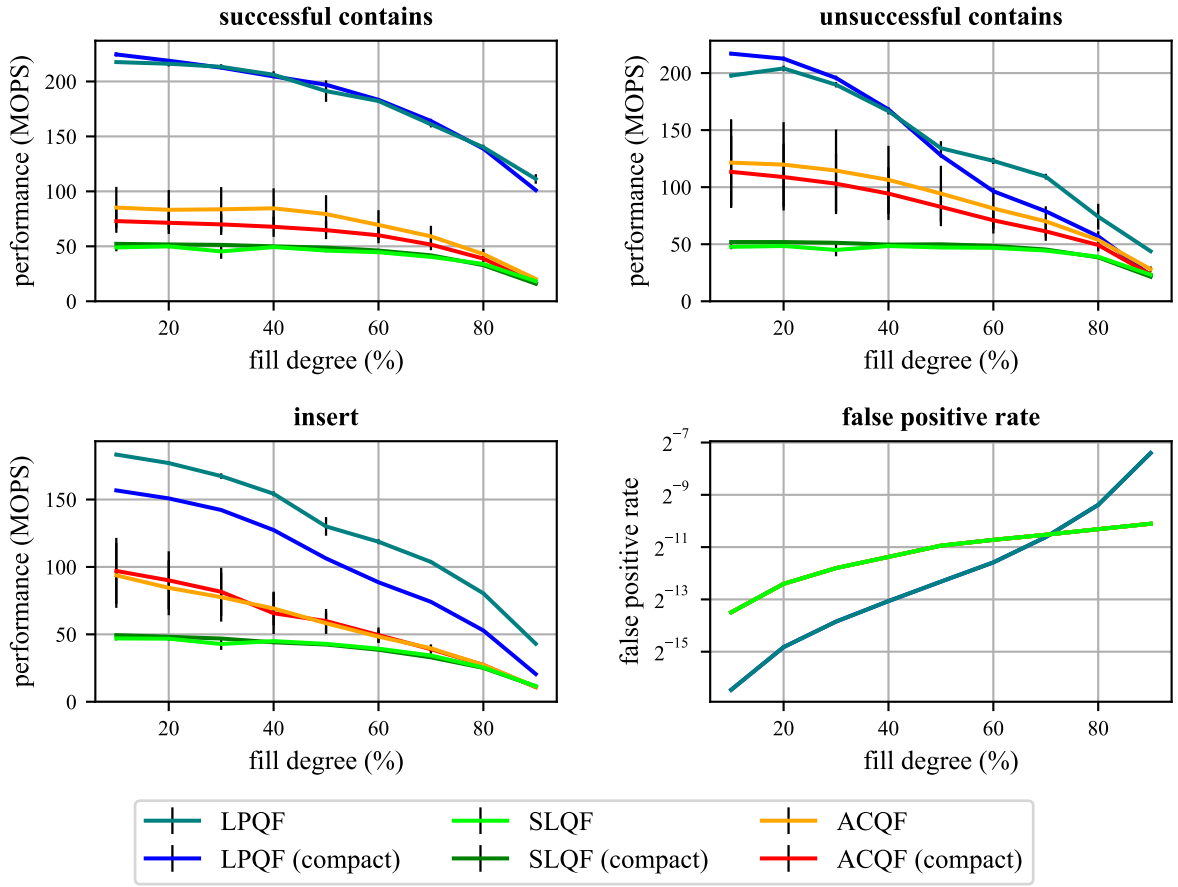


Figure 6.2: Measuring the performance of different concurrent quotient filters for varying fill degrees. In the false positive rate plot the blue line includes both LPQF variants while the green line includes the remaining filters. The used parameters are: $m = 2^{26}$, $\varepsilon = 2^{-10}$, test size = 100 000.

as for an empty filter. At 90% fill degree both the LPQF and ACQF lose their performance advantages as all filters have a nearly identical performance for these two operations.

The false positive rate of the ACQF and SLQF shows the expected behavior of growing asymptotically towards the maximum false positive rate of $\varepsilon = 2^{-10}$ for these experiments. The LPQF starts with an lower initial false positive rate but also grows faster with increasing fill degree. At 70% fill degree it equals the other two filters and at 75% fill degree it surpasses the given maximum false positive rate even though it uses three additional remainder bits compared to the other two QFs.

6.2.3 Influence of Fill Degree on Efficiency

Figure 6.3 shows the influence of the number of threads and the fill degree on the efficiency of the ACQF. Efficiency is the relative speedup divided by the thread count and measures how well each thread utilizes its resources. We can see that adding more threads decreases the efficiency as is to be expected. This decrease remains constant over varying fill degrees. We can also see slightly larger decrease in efficiency when using the second socket for more than 8 threads and when using hyper threading for more than 16 threads.

The measurements show that the operations behave slightly different over the tested fill degree range of 10% to 90% . The efficiency of the successful contains operation steadily increases with larger fill degrees with a total increase of 0.25 on average. The efficiency of successful contains operation initially decreases slightly until a fill degree of 50% is reached. From this point onwards, it increases again, giving an average total increase of about 0.15. For insert operations the efficiency increases until a maximum at about 70% fill degree is reached at which we have an efficiency increase of about 0.15 on average. For even higher fill degrees the efficiency decreases again by about 0.05. Note that for higher thread counts the efficiency tends to increase slightly more with larger fill degrees than for lower thread counts.

The reason for the increase of efficiency with larger fill degrees for the ACQF is due to the amount of locking overhead relative to the remaining work of the operation. For small fill degrees any operation on a sequential QF is very fast as the cluster lengths are very small on average and only a few entries have to be read or modified. Here the performance impact of locking is very noticeable as acquiring locks can take longer than the time needed to complete the remaining insert or contains operation. Clusters and super clusters get longer for larger fill degrees which increases the average work per operation. Therefore, the relative performance impact of acquiring locks goes down. This results in an overall increase of efficiency.

The false positive rate for the ACQF is independent of the number of threads used and behaves just as in a sequential quotient filter.

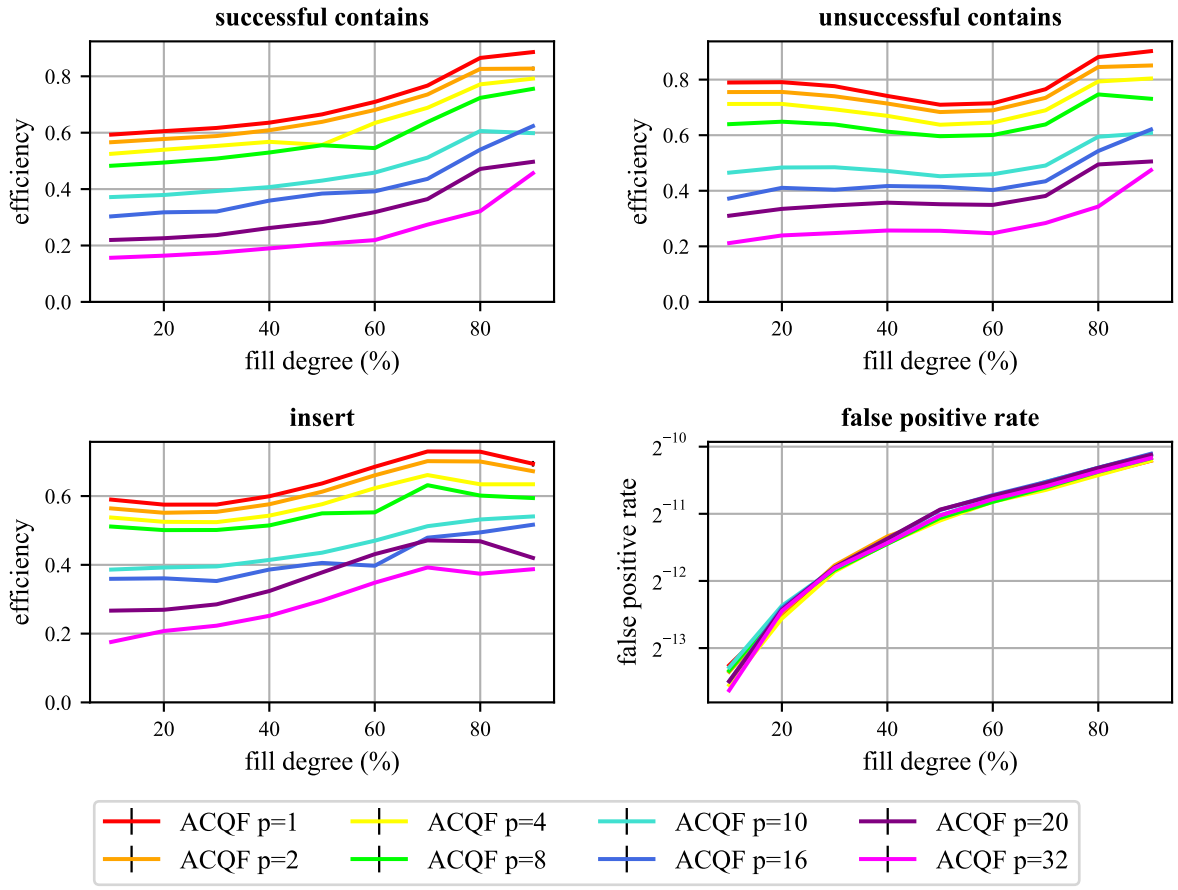


Figure 6.3: Measuring the efficiency of the ACQF relative to a sequential quotient filter with different number of threads (p) over varying fill degrees. Efficiency is the relative speedup divided by p. The used parameters are: $m = 2^{26}$, $\varepsilon = 2^{-10}$, test size $= 100\,000$.

6.2.4 Remainder Bits

In this section we look at the effect of the number of remainder bits on the performance of the different operations. In figure 6.4 we can see that the LPQF remains consistently faster than the ACQF which is faster than the SLQF.

Note that we do not add three extra remainder bits for the LPQF for these measurements. This is why the LPQF has a higher false positive rate than the ACQF and SLQF.

Using the compact version of the LPQF yields a big gain in performance when using only a small number of remainder bits. The compact ACQF also yields performance improvements over its non-compact counter part for insert operations. The performance of the SLQF is not changed by using the compact representation since the locking overhead overshadows any positive or negative impact on performance. If we use more than 5 remainder bits the relative performance differences between the quotient filters remain almost constant and very similar to the results presented in 6.2.1. The difference in performance increase is due to the different locking overheads which reduce the benefits from the compact representation.

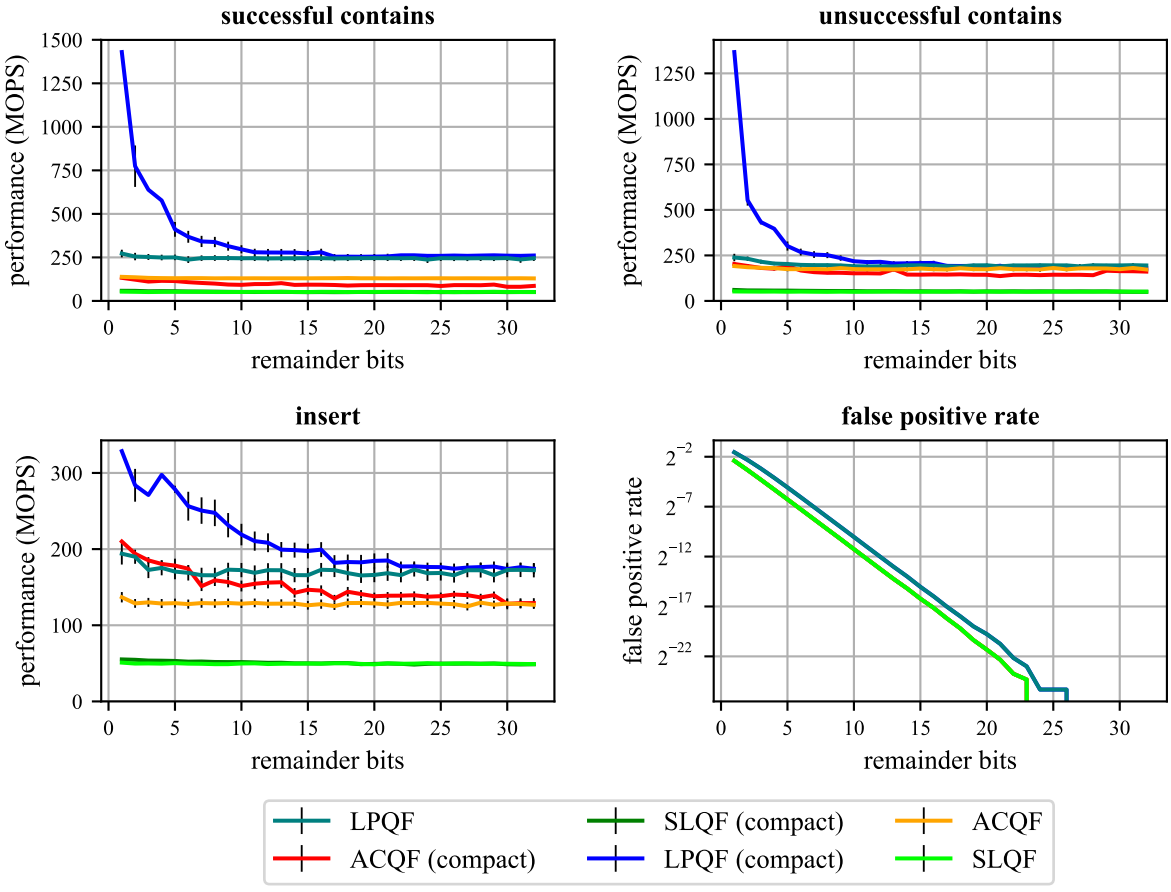


Figure 6.4: Measuring the performance of different concurrent quotient filters with remainder bits in the range $[1,32]$. In the false positive rate plot the blue line includes both LPQF variants while the green line includes the remaining filters. The used parameters are: $n = 14000000$, $m = 2^{25}$.

6.2.5 Mixed Reads and Writes

Insert and contains operations are allowed to happen concurrently for all presented concurrent filter variants. Figure 6.5 shows the effect of different rations of read and write operations. The read operations are composed of equal parts successful and unsuccessful contains operations. We first fill the given filter variant to 40% of its maximum capacity. Then we perform the concurrent operations such that each thread performs the same amount of operations in which each insert is followed by a number of contains operations determined by the given read-write ratio. The contains operations are alternating between successful and unsuccessful contains operations. The insert operations account for a total increase in fill degree of 10%.

We can see that the performance of the ACQF and SLQF remains constant with different read-write ratios whereas it can vary for the LPQF. For ratios above 3 the compact variant is slightly faster for the LPQF and slightly slower for the ACQF. Overall the LPQF is about two times faster on average than the ACQF which is itself two times faster than the SLQF due to the different locking overheads.

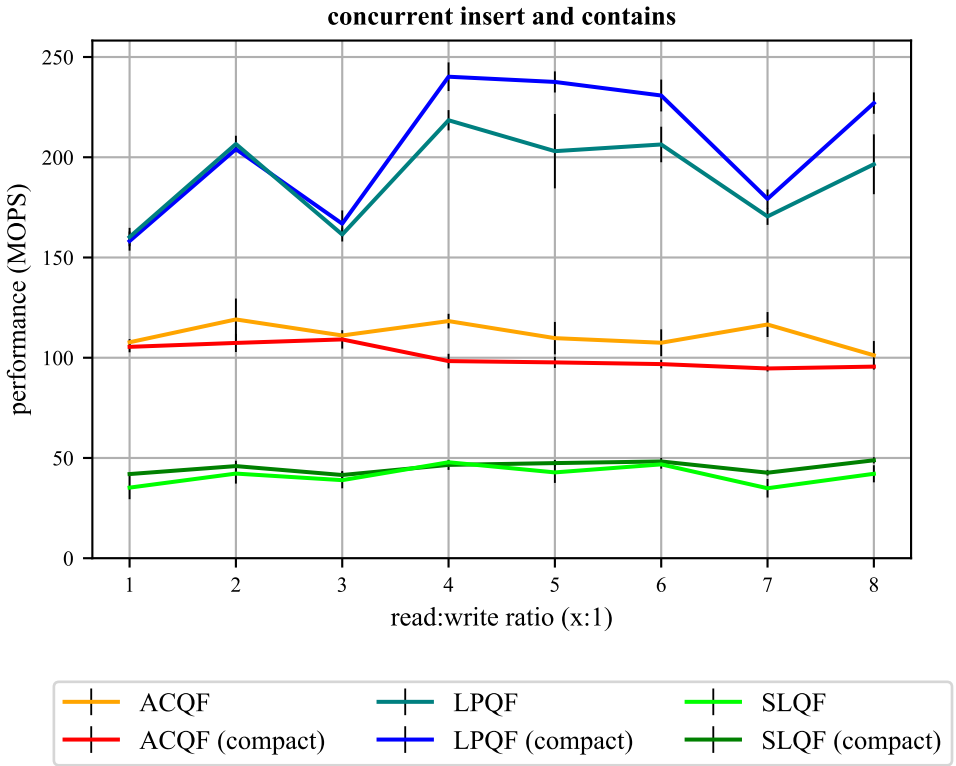


Figure 6.5: Measuring the performance of concurrent insert and contains operations of different concurrent quotient filters with varying read to write ratios. The used parameters are: $n = 14\,000\,000$, $m = 2^{25}$, $\varepsilon = 2^{-10}$.

6.3 Growing and Dynamic Quotient Filters

In the second part of the experimental evaluation we look at the bounded growing quotient filter (BGQF) as well as the fully dynamic quotient filter (FDQF) presented in section 5. Both are evaluated with and without compact arbitrary length data types. We compare them against each other and against a non-growing quotient filters. We also evaluate their speedup and efficiency when using multiple threads. We look at their memory usage and at the impact of using quick inserts with the FDQF. We do not use hazard pointers to remove the quotient filters that are no longer used after they have grown during these experiments. Instead we explicitly look at their overhead in section 6.3.6.

6.3.1 Overhead of Growing Quotient Filters

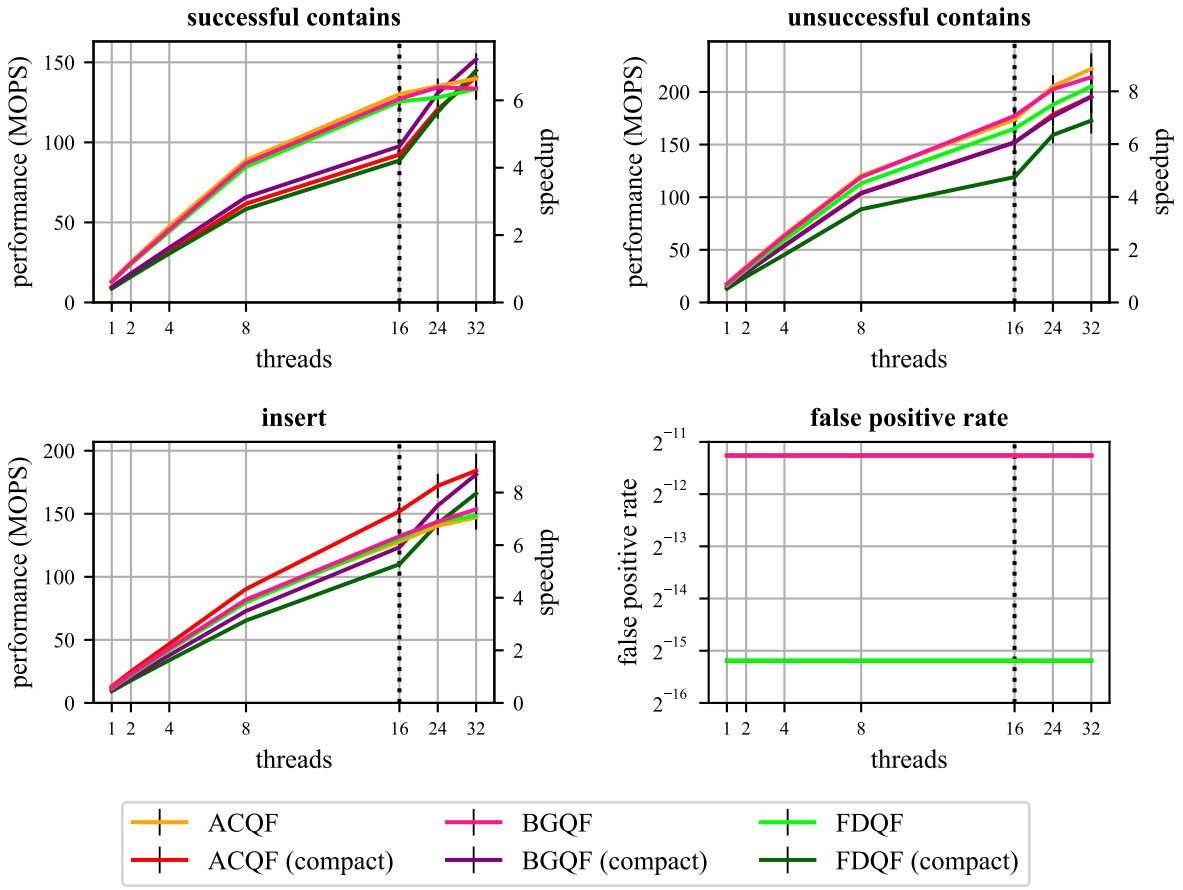


Figure 6.6: Measuring the performance overhead of different growing quotient filters compared to the ACQF and the speedup relative to a sequential quotient filter with varying number of threads. The dashed line indicates that hyper threading is used for measurements with 24 and 32 threads (note the different x-axis scale). In the false positive rate plot the green line includes both FDQF variants while the pink line includes the remaining filters. The used parameters are: $n = 140\,000\,000$, $m = 2^{25}$, $\varepsilon = 2^{-10}$.

In this section we show the performance overhead of the FDQF and BGQF compared to a non-growing ACQF. All filters are constructed with the same initial capacity and we fill the quotient filters such that no growing occurs so that we can measure the overhead that the FDQF and BGQF have compared to the ACQF. We first fill the filters to 40% of their capacity and then do the measurements of the contains operations just as in section 6.2.1.

As we can see in figure 6.6 the FDQF and BGQF, which are in essence just wrapper data structures around one or more ACQF, have very little overhead and scale just as well with additional threads as the non-growing quotient filters. The largest variation in performance can be seen for insert and unsuccessful contains operations using 16 threads where the performance of the compact FDQF is about 30% smaller than that of the compact ACQF. In all other cases the growing quotient filters are at most 10% slower than the ACQF counterpart. The lower false positive rate of the FDQF is a result of how the maximum false positive rates of the different levels have to be chosen to guarantee an overall false positive rate of the FDQF that is below the limit set by the user. Since we want to avoid growing in this experiment, only the first level of the FDQF is used.

6.3.2 Impact of Growing Operations

The impact of growing operations on performance in comparison to the non-growing ACQF can be seen in figure 6.7. The ACQF is constructed with a maximum capacity of 2^{25} while the FDQF and BGQF are constructed with an initial capacity of 2^{17}. This means that during the insertion phase where we fill the filters with 14 000 000 elements the FDQF and BGQF have to grow multiple times. Afterwards we perform the measurements of the contains operations just as in section 6.2.1.

We see that the additional growing operations during the insertions add a large overhead and decrease performance significantly compared to not performing any growing operations as seen in 6.3.1. The performance of the BGQF for the contains operations are not changed by growing during insertions since there is always only one active quotient filter where all entries are stored. The FDQF on the other hand constructs new levels during growing operations which means that contains operations also have to potentially check multiple quotient filters for the presence of an element. Therefore, the FDQF is also slower for contains operations compared to a FDQF with only one level containing the same elements.

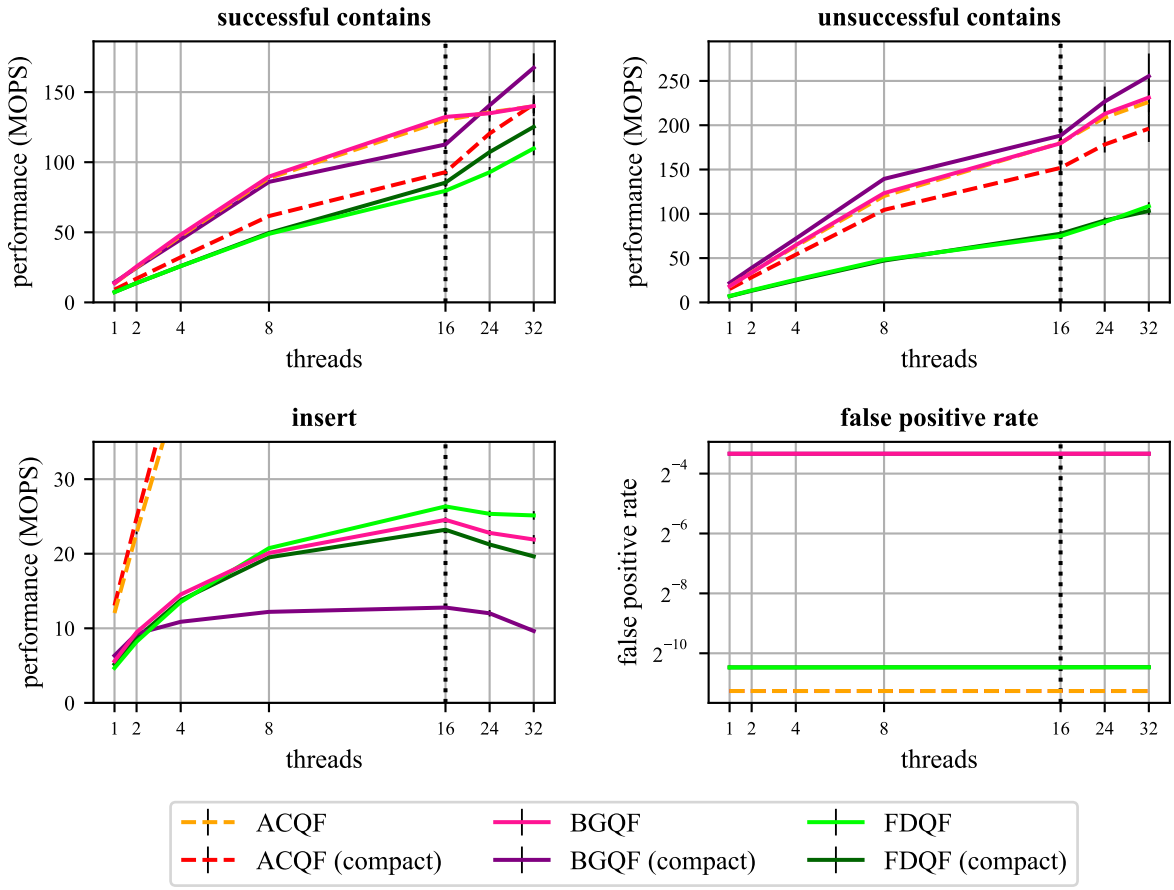


Figure 6.7: Measuring the performance of different growing quotient filters compared to the ACQF and the speedup relative to a sequential quotient filter with varying number of threads. The growing quotient filters perform multiple growing operations during the insertions. The black dashed line indicates that hyper threading is used for measurements with 24 and 32 threads (note the different x-axis scale). In the false positive rate plot each line includes the respective compact and non-compact variant. The used parameters are: $n = 14\,000\,000$, $m_{ACQF} = 2^{25}$, initial capacity for growing QFs $= 2^{17}$, $\varepsilon = 2^{-10}$.

6.3.3 Fill Degree

The fill degree of a quotient filter affects its performance as shown in 6.2.2. This is also true for BGQFs and FDQFs even though they have the ability to grow as can be seen in figure 6.8. For this measurement we split the total number of elements which will be inserted into 50 equal parts and measure the insertion time of every part. After each insertion we also measure the running time and false positive rate for both contains operations.

Since both tested filters can grow, the fill degree of any internally used filter of the BGQF and FDQF remains below its maximum fill degree $\alpha_{\max}$ because this is the point at which

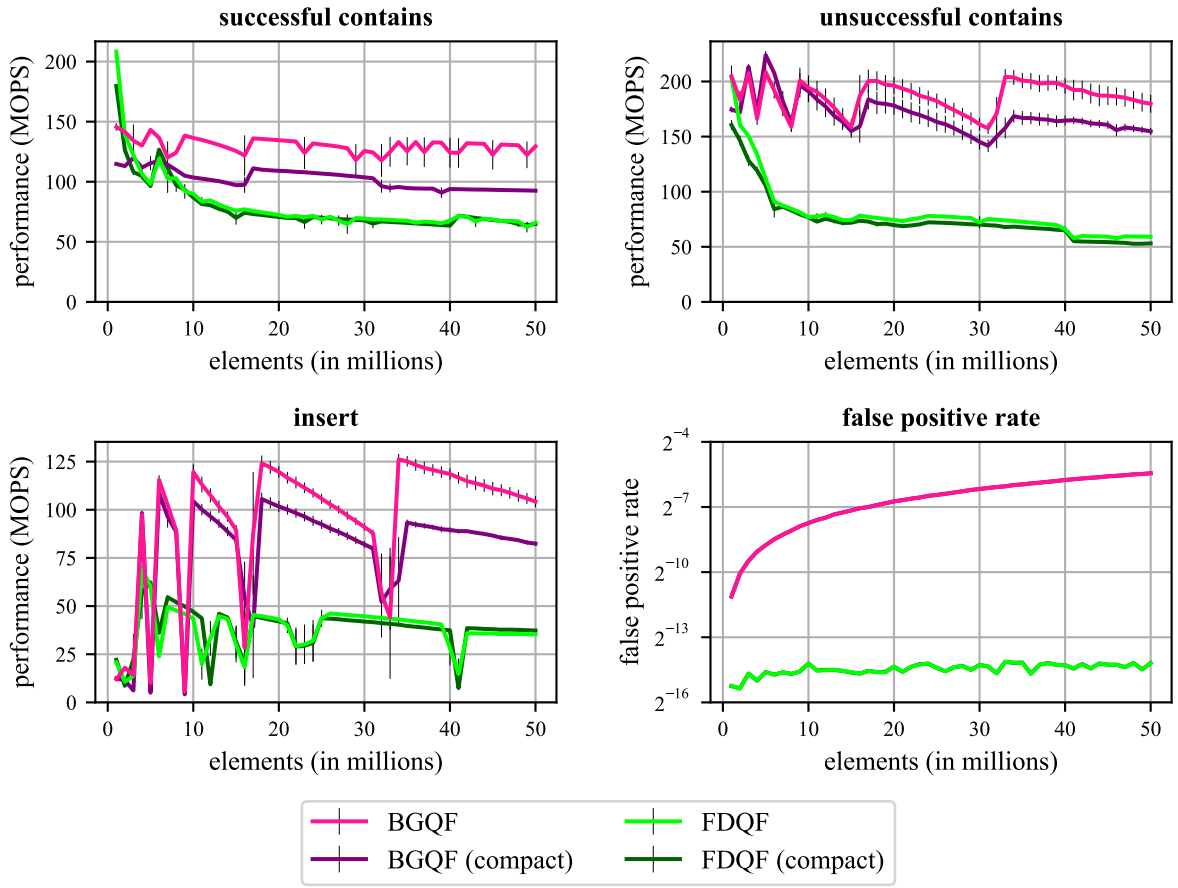


Figure 6.8: Measuring the performance of different growing quotient filters for a varying number of inserted elements. The false positive rates for the respective compact and non-compact counterparts are the same. The used parameters are: $n = 50000000$, initial capacity $= 2^{17}$, $\varepsilon = 2^{-14}$.

the filters will grow. Thus the performance drops slightly as the internal filter approaches $\alpha_{\max}$ and then suddenly increases again after growing occurred. The growing operation has to linearly traverse the array of a QF and can have a large impact on insert performance.

The FDQF has lower performance than the BGQF for every operation as soon as a second level is added. On average the BGQF is 1.5 times faster for successful contains, 3 times faster for unsuccessful contains and 2 times faster for insertions. We can also see that the performance of the BGQF varies with its fill degree while the FDQF has more stable performance values because the accesses are distributed across multiple levels. Using the compact representation decreases the performance of the BGQF by up to 25% on average whereas it has no effect on the performance of the FDQF. The false positive rate of the FDQF remains strictly below the maximum false positive rate set through user input whereas the BGQF quickly goes beyond this threshold.

6.3.4 Memory Usage

In figure 6.9 we can see how efficient the allocated memory of the FDQF and BGQF is used. We measure the number of allocated bits immediately before and after a grow operation takes place. This includes adding a new level to the FDQF as well as growing the BGQF. The tested FDQF uses a BGQF for each level which can grow three times until it reaches its maximum capacity. If the size of one entry does not divide the integer it is stored in exactly then there is some amount of bits in each integer that is unused even when using the compact representation. In order to see the impact of these unused bits we also added theoretically optimal curves for the FDQF and BGQF where the integers storing the entries are exactly the size of one entry so a fully filled QF would not have any unused bits.

We can see that the non-compact variants use between three and seven times as much memory to store the same number of elements on average than their compact counterparts which are in turn very close to the theoretical optimum. Growing a quotient filter allocates new memory but does not change the number of elements in the filters which results in a sharp increase in allocated bits per element. This behavior causes the spikes which can be seen for every curve. We can see that the first four spikes occur at the same number of elements for both the FDQF and the BGQF. That is because each level of the FDQF is also a BGQF so when we only have one level in the fully dynamic quotient filter it behaves the same as a normal BGQF. This changes when this first level reaches its maximum grow steps and instead of growing further a new level is added. Since the newly added level is initially smaller than the filled previous levels it reaches it next growing point faster but it does not need to allocate as much new memory as the BGQF. This is why we see more but smaller spikes for the FDQF for the non-compact variants.

The compact variant of the BGQF uses the least amount of allocated bits per elements and is very close to an optimal BGQF. But the cost of using a BGQF over an FDQF is that the false positive rate grows steadily towards 100% and can not be bounded by the user. In contrast the FDQF asymptotically grows towards its maximum false positive rate.

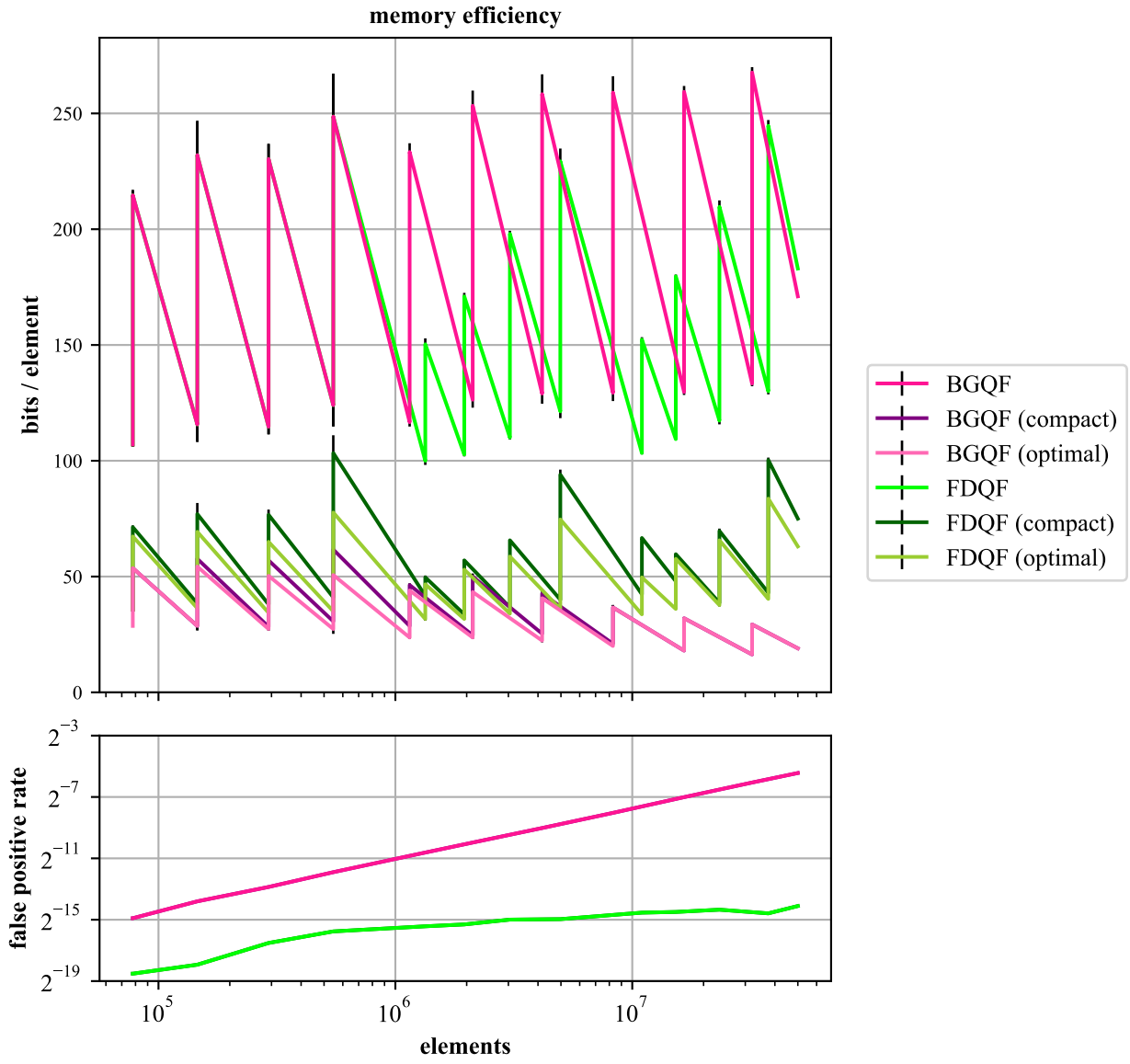


Figure 6.9: Measuring the memory usage per element of different growing quotient filters for a varying number of inserted elements. The variants annotated with (*optimal*) show the theoretical memory usage for filters where the integers of the internal array match the size of one entry exactly. The false positive rates for the respective compact and non-compact counterparts are the same. The used parameters are: $n = 50\,000\,000$, initial capacity $= 2^{17}$, $\varepsilon = 2^{-14}$.

6.3.5 Quick Insert

In this section we evaluate the benefit and shortcomings of using quick inserts with the FDQF. As discussed in 5.3, each level of the FDQF has a maximum fill degree $\alpha_{\max}$ which we set to 0.6 for this experiment. It sets the limit on how many elements can be inserted into a level and when a new level is added. Through the use of quick inserts we can still gradually fill the remaining empty slots with new entries. Each entry which is inserted by a quick insert is not inserted into the active level and therefore delays the creation of further levels. This can be seen in figure 6.10 which also shows that until the second level is added the FDQFs have exactly the same memory usage. The false positive rate is slightly higher when using quick inserts but the user defined overall maximum false positive rate for the FDQF is still a strict upper bound.

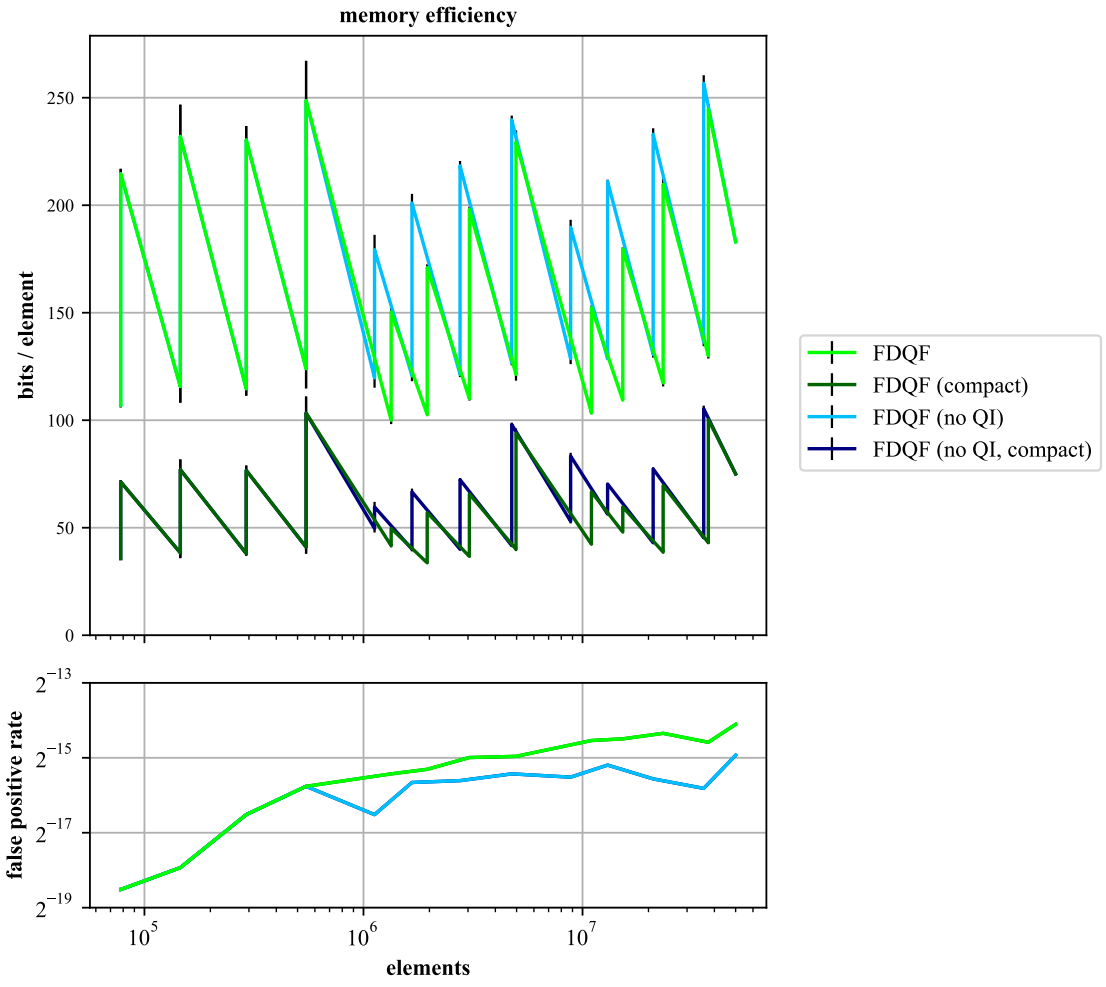


Figure 6.10: Measuring the memory usage per element of the FDQF with and without quick inserts (*QI*) for a varying number of inserted elements. The false positive rates for the respective compact and non-compact counterparts are the same. The used parameters are: $n = 50\,000\,000$, initial capacity $= 2^{17}$, $\varepsilon = 2^{-14}$.

Figure 6.11 shows the performance difference when using quick inserts with FDQFs. We can see that inserts are about 50% slower because we need to try every level in order until either a quick insert succeeds or we reach the active level where we perform a regular insert operation. Both types of contains operations are about 25% faster when using quick inserts. The reason for this is that unlike insert operations contains operations always have to traverse all levels in order until the element is found or the last level is reached. Because of quick inserts more elements are inserted into the first levels of the FDQF which is why successful contains operations are more likely to find the given element earlier. Unsuccessful contains operations benefit from additional optimizations as a result of using quick inserts which also allows the to finish before visiting all levels.

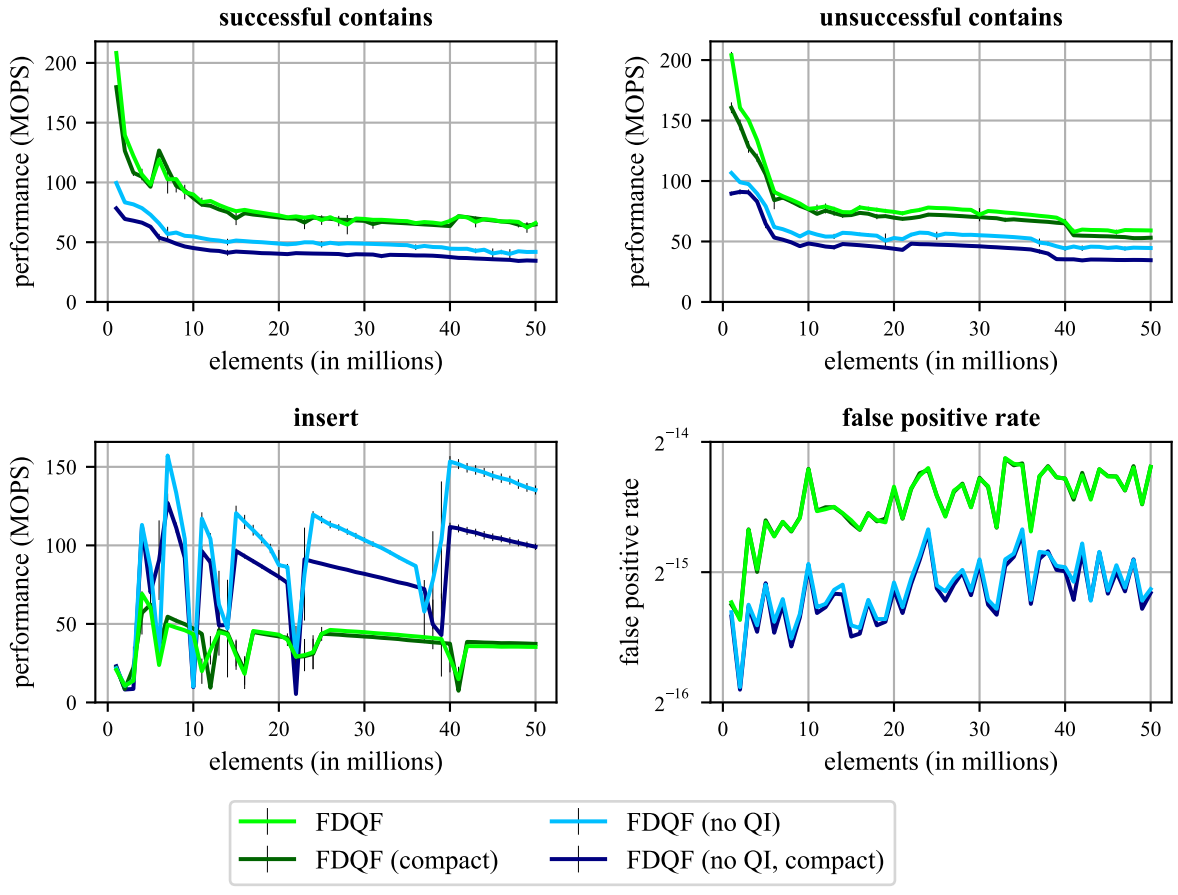


Figure 6.11: Measuring the performance of the FDQF with and without quick inserts for a varying number of inserted elements. In the false positive rate plot the green line includes both FDQF variants. The used parameters are: $n = 50000000$, initial capacity $= 2^{17}$, $\varepsilon = 2^{-14}$.

6.3.6 Overhead of Hazard Pointers

In this section we evaluate the impact of using hazard pointers for growing quotient filters. Hazard pointers are a way of safely reclaiming the memory of quotient filters that are no longer used after they have grown and their entries have been moved to a bigger quotient filter. We use the same measurement procedure as in 6.3.1 where we measure the overhead of growing operations during insertions.

We see in figure 6.12 that hazard pointers slow the performance of the FDQF and BGQF by a factor of 2 on average. This results from the additional atomic operations and cache misses that are necessary in order to protect and release the hazard pointers.

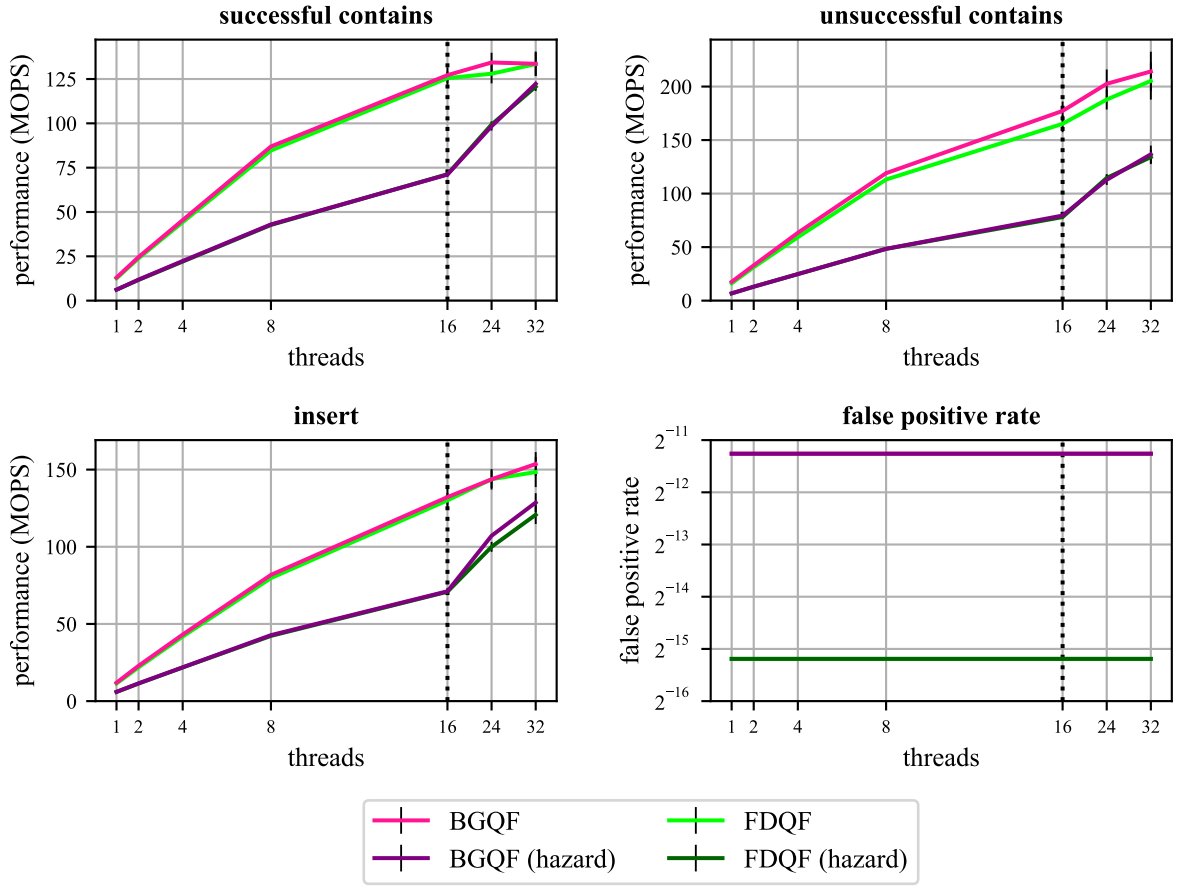


Figure 6.12: Measuring the performance overhead of using hazard pointers with different growing quotient filters with varying number of threads. The dashed line indicates that hyper threading is used for measurements with 24 and 32 threads (note the different x-axis scale). In the false positive rate plot the green line includes both FDQF variants while the purple line includes the remaining filters. The used parameters are: $n = 14000000$, $m = 2^{25}$, $\varepsilon = 2^{-10}$.

7 Conclusion

In this thesis we presented different improvements for standard quotient filters. By using arbitrary compact data types to store the entries of the quotient filter we were able to almost completely avoid unused allocated memory while having only slight positive or negative impact on performance, depending on the usage scenario. We show different techniques for implementing concurrent quotient filters which are up to four times faster than traditional locking approaches. With the ACQF we showed how to use the inherent structure of quotient filters to implement local locking and we also looked at the LPQF which trades strict upper bounds on the false positive rate for better performance and speedup. Finally, with the fully dynamic quotient filter we looked at how to build a quotient filter that is able to dynamically grow and still enforces a strict upper bound on its false positive rate while being no more than 10% slower as non-growing quotient filters in scenarios where growing is not necessary.

These improvements make quotient filters a very useful AMQ data structure not only in theory but also in many practical situations. Since all aforementioned improvements – compact storage, concurrency and dynamic growing – can be freely combined with each other, this gives us a flexible set of AMQ data structures based on quotient filters where users can easily choose the best fit for their particular problem.

7.1 Future Work

The multilevel structure of the fully dynamic quotient filter can be further explored by using different growing and non-growing quotient filters as levels. It may even be possible to use this approach with filters similar to the presented LPQF and still be able to achieve a strict upper bound on the false positive rate.

The compact representation can be applied to non-standard quotient filters like the counting quotient filter in order to improve their memory efficiency. The presented localized locking mechanism can also be used to parallelized other quotient filters and can possibly be extended to also allow for concurrent remove operations.

Bibliography

[1] M. Al-hisnawi and M. Ahmadi. Deep packet inspection using quotient filter. *IEEE Communications Letters*, 20(11):2217–2220, Nov 2016.

[2] Paulo Sérgio Almeida, Carlos Baquero, Nuno Preguiça, and David Hutchison. Scalable bloom filters. *Inf. Process. Lett.*, 101(6):255–261, March 2007.

[3] Michael A Bender, Martin Farach-Colton, Rob Johnson, Russell Kraner, Bradley C Kuszmaul, Dzejla Medjedovic, Pablo Montes, Pradeep Shetty, Richard P Spillane, and Erez Zadok. Don't thrash: how to cache your hash on flash. *Proceedings of the VLDB Endowment*, 5(11):1627–1637, 2012.

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

[5] Pedro Celis. *Robin Hood Hashing*. PhD thesis, Waterloo, Ont., Canada, Canada, 1986.

[6] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. Cuckoo filter: Practically better than bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, pages 75–88, New York, NY, USA, 2014. ACM.

[7] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. In *Proceedings of the ACM SIGCOMM '98 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '98, pages 254–265, New York, NY, USA, 1998. ACM.

[8] Dimitris Fotakis, Rasmus Pagh, Peter Sanders, and Paul Spirakis. Space efficient hash tables with worst case constant access time. *Theory of Computing Systems*, 38(2):229–248, Feb 2005.

[9] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[10] Tobias Maier and Peter Sanders. Dynamic space efficient hashing. *CoRR*, abs/1705.00997, 2017.

[11] Michael Mitzenmacher, Salvatore Pontarelli, and Pedro Reviriego. Adaptive cuckoo filters. In *2018 Proceedings of the Twentieth Workshop on Algorithm Engineering and Experiments (ALENEX)*, pages 36–47. SIAM, 2018.

[12] Prashant Pandey, Fatemeh Almodaresi, Michael A. Bender, Michael Ferdman, Rob Johnson, and Rob Patro. Mantis: A fast, small, and exact large-scale sequence search index. *bioRxiv*, 2017.

[13] Prashant Pandey, Michael A. Bender, Rob Johnson, and Rob Patro. A general-purpose counting filter: Making every bit count. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, pages 775–787, New York, NY, USA, 2017. ACM.

[14] Jarosław Piskorski and Przemysław Guzik. Filtering poincare plots. *Computational methods in science and technology*, 11(1):39–48, 2005.

[15] Felix Putze, Peter Sanders, and Johannes Singler. Cache-, hash-, and space-efficient bloom filters. *J. Exp. Algorithmics*, 14:4:4.4–4:4.18, January 2010.