Concurrent Expandable AMQs on the Basis of Quotient Filters

Tobias Maier
Karlsruhe Institute of Technology, Germany
t.maier@kit.edu

Peter Sanders
Karlsruhe Institute of Technology, Germany
sanders@kit.edu

Robert Williger
Karlsruhe Institute of Technology, Germany

Abstract

A quotient filter is a cache efficient Approximate Membership Query (AMQ) data structure. Depending on the fill degree of the filter most insertions and queries only need to access one or two consecutive cache lines. This makes quotient filters very fast compared to the more commonly used Bloom filters that incur multiple independent memory accesses depending on the false positive rate. However, concurrent Bloom filters are easy to implement and can be implemented lock-free while concurrent quotient filters are not as simple. Usually concurrent quotient filters work by using an external array of locks – each protecting a region of the table. Accessing this array incurs one additional memory access per operation. We propose a new locking scheme that has no memory overhead. Using this new locking scheme we achieve 1.6× times higher insertion performance and over 2.1× higher query performance than with the common external locking scheme.

Another advantage of quotient filters over Bloom filters is that a quotient filter can change its capacity when it is becoming full. We implement this growing technique for our concurrent quotient filters and adapt it in a way that allows unbounded growing while keeping a bounded false positive rate. We call the resulting data structure a fully expandable quotient filter. Its design is similar to scalable Bloom filters, but we exploit some concepts inherent to quotient filters to improve the space efficiency and the query speed.

Additionally, we propose several quotient filter variants that are aimed to reduce the number of status bits (2-status-bit variant) or to simplify concurrent implementations (linear probing quotient filter). The linear probing quotient filter even leads to a lock-free concurrent filter implementation. This is especially interesting, since we show that any lock-free implementation of other common quotient filter variants would incur significant overheads in the form of additional data fields or multiple passes over the accessed data. The code produced as part of this submission can be found at `https://www.github.com/Toobiased/lpqfilter`.

2012 ACM Subject Classification Theory of computation → Data structures design and analysis; Theory of computation → Shared memory algorithms

Keywords and phrases Quotient filter, Concurrent data structures, Locking

Digital Object Identifier 10.4230/LIPIcs.SEA.2020.15

Supplementary Material The code produced as part of this submission can be found at `https://www.github.com/Toobiased/lpqfilter`.

1 Introduction

Approximate Membership Query (AMQ) data structures offer a simple interface to represent sets. Elements can be inserted, queried, and depending on the use case elements can also be removed. A query returns true if the element was previously inserted. Querying an element might return true even if it was not inserted. We call this a false positive. The probability that a non-inserted element leads to a false positive is called the false positive rate of the

filter. It can usually be chosen when constructing the data structure. AMQ data structures have two main advantages over other set representations they are fast and space efficient. Similar to hash tables most AMQ data structures have to be initialized with their capacity. This can be a problem for their space efficiency when the final size is unknown a priori.

AMQ data structures have become an integral part of many complex data structures or data base applications. Their small size and fast accesses can be used to sketch large, slow data sets. The AMQ filter is used before accessing the data base to check whether a slow lookup is actually necessary. Some recent uses of AMQs include network analysis [1] and bio informatics [15]. These are two of the areas with the largest data sets available today, but given the current big data revolution we expect more and more research areas will have a need for the space efficiency and speedup potential of AMQs. The most common AMQ data structure in practice is still a Bloom filter even though a quotient filter would usually be significantly faster. One reason for this is probably inertia, but another reason might be that concurrent Bloom filters are easy to implement – even lock-free and well scaling implementations. This is important because well scaling implementations have become critical in today's multi-processor scenarios since single core performance stagnates and efficient multi-core computation has become a necessity to handle growing data sets.

We present a technique for concurrent quotient filters that uses fine grained local locking inside the table – without an external array of locks. Instead of traditional locks we use the per-slot status bits that are inherent in quotient filters to lock local ranges of slots. As long as there is no contention on a single area of the table there is very little overhead to this locking technique because the locks are placed in slots that would have been accessed anyways. The capacity of a quotient filter can be increased without access to the original elements. However, because the false positive rate grows linearly with the number of inserted elements, this is only useful for a limited total growing factor. We implement this growing technique for our concurrent quotient filters and extend it to allow large growing factors at a strictly bounded false positive rate. These *fully expandable quotient filters* combine growing quotient filters with the multi level approach of scalable Bloom filters [2].

Related Work

Since quotient filters were first described in 2012 [3] there has been a steady stream of improvements. For example Pandey et al. [16] have shown how to reduce the memory overhead of quotient filters by using rank-select data structures. This also improves the performance when the table becomes full. Additionally, they show an idea that saves memory when insertions are skewed (some elements are inserted many times). They also mention the possibility for concurrent access using an external array of locks (see Section 5 for results). Recently, there was also a GPU-based implementation of quotient filters [9], further indicating that there is a lot of interest in concurrent AMQs even in these highly parallel scenarios.

Quotient filters are not the only AMQ data structures that have received attention recently. Cuckoo filters [8, 13] and very recently Morton filters [4] (based on cuckoo filters) are two other examples of AMQ data structures. Due to their similarity to cuckoo hash tables, they do not lend themselves to easy parallel solutions (insertions can have large effects on the overall data structure).

A scalable Bloom filter [2] allows unbounded growing by adding additional levels of Bloom filters once a level becomes full. Each new filter is initialized with a higher capacity and more hash functions than the last. The query time is dependent on the number of times the filter has grown both because more filters have to be checked and because later filters have more hash functions. In Section 4, we show a similar technique for fully expandable quotient filters that mitigates many of these problems.

Overview. In Section 2 we introduce the terminology, sequential quotient filters as well as our filter variations in a sequential setting. Section 3 describes our approach to concurrent quotient filters the lock-free linear probing quotient filter and the locally locked quotient filter. The dynamically growing quotient filter variant is described in Section 4. All presented data structures are evaluated in Section 5. Afterwards, we draw our conclusions in Section 6.

2 Sequential Quotient Filter

In this section we describe the basic sequential quotient filter as well as some variants to the main data structure. Throughout this paper, we use m for the number of slots in our data structure and n for the number of inserted elements. The fill degree of any given table is denoted by $\delta = n/m$. Additionally, we use p^+ to denote the probability of a false positive query. Quotient filters are approximate membership query data structures that were first described by Bender et al. [3] and build on an idea for space efficient hashing originally described by Cleary [6]. Quotient filters replace possibly large elements by *fingerprints*. The fingerprint $f(x)$ of an element x is a number in a predefined range $f: x \mapsto \{0, ..., 2^k - 1\}$ (binary representation with exactly k digits). We commonly get a fingerprint of x by taking the k bottommost bits of a hash function value $h(x)$ (i.e., xxHash [7]).

A quotient filter stores the fingerprints of all inserted elements. When executing a query for an element x, the filter returns `true` if the fingerprint $f(x)$ was previously inserted and `false` otherwise. Thus, a query looking for an element that was inserted always returns `true`. A false positive occurs when x was not inserted, but its fingerprint $f(x)$ matches that of a previously inserted element. Given a fully random fingerprint function, the probability of two fingerprints being the same is 2^{-k}. Therefore, the probability of a false positive is bounded by $n \cdot 2^{-k}$ where n is the number of stored fingerprints.

To achieve expected constant query times as well as to save memory, fingerprints are stored in a special data structure that is similar to a hash table with open addressing (specifically it is similar to robinhood hashing [5]). During this process, the fingerprint of an element x is split into two parts: the topmost q bits called the quotient $quot(x)$ and the bottommost r bits called the remainder $rem(x)$ ($q + r = k$). The quotient is used to address a table that consisting of $m = 2^q$ memory slots of $r + 3$ bits (a slot can store one remainder and three additional status bits). The quotient of each element is only stored implicitly by the position of the element in the table. The remainder is stored explicitly within one slot of the table. Similar to many hashing techniques, we try to store each element in one designated slot (index $quot(x)$) which we call its canonical slot. With the help of the status bits we can reconstruct quotient of each stored element even when it is shifted after its canonical slot.

Table 1 Meaning of different status bit combinations.

`1**`	this slot has a run
`000`	empty slot
`100`	cluster start
`*01`	run start
`*11`	continuation of a run
`*10`	– (not used see 3.2)

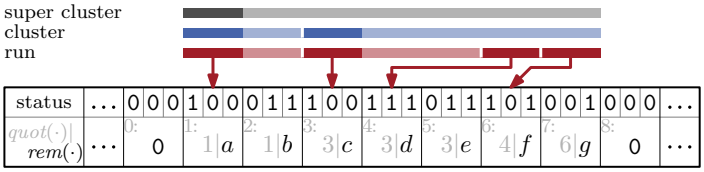

Figure 1 Section of the table with highlighted runs, clusters, and superclusters. Runs point to their canonical slot.

The main idea for resolving collisions is to find the next free slot – similar to linear probing hash tables – but to reorder the elements such that they are sorted by their fingerprints (see Figure 1). Elements with the same quotient (the same canonical slot) are stored in

consecutive slots, we call them a run. The canonical run of an element is the run associated with its canonical slot. The canonical run of an element does not necessarily start in its canonical slot. It can be shifted by other runs. If a run starts in its canonical slot we say that it starts a cluster that contains all shifted runs after it. Multiple clusters that have no free slots between them form a supercluster. We use the 3 status bits that are part of each slot to distinguish between runs, clusters, and empty slots. For this we store the following information about the contents of the slot (further described in Table 1): Were elements hashed to this slot (is its run non-empty)? Does the element in this slot belong to the same run as the previous entry (used as a run-delimiter signaling where a new run starts)? Does the element in this slot belong to the same cluster as the previous entry (is it shifted)?

During the query for an element x, all remainders stored in x's canonical run have to be compared to $rem(x)$. First we look at the first status bit of the canonical slot. If this status bit is unset there is no run for this slot and we can return false. If there is a canonical run, we use the status bits to find it by iterating to the left until the start of a cluster (third status bit $= 0$). From there, we move to the right counting the number of non-empty runs to the left of the canonical run (slots with first status bit $= 1$ left of $quot(x)$) and the number of run starts (slots with second status bit $= 0$). This way, we can easily find the correct run and compare the appropriate remainders.

An insert operation for element x proceeds similar to a query, until it either finds a free slot or a slot that contains a fingerprint that is $\geq f(x)$. The current slot is replaced with $rem(x)$ shifting the following slots to the right (updating the status bits appropriately).

When the table fills up, operations become slow because the average cluster length increases. When the table is full, no more insertions are possible. In this case, quotient filters can be migrated into a larger table – increasing the overall capacity. To do this a new table is allocated with twice the size of the original table and one less bit per remainder. Addressing the new table demands an additional quotient bit, since it is twice the size. This issue is solved by moving the uppermost bit from the remainder into the quotient ($q' = q + 1$ and $r' = r - 1$). The fingerprint size and number of elements remain the same, therefore, the capacity increase does not impact the false positive rate of the filter. But the false positive rate still increases linearly with the number of insertions ($p^+ = n \cdot 2^{-k}$). Therefore, the false positive rate doubles when the table is filled again. We call this migration technique *bounded growing*, because to guarantee reasonable false positive rates this method of growing should only be used a bounded number of times.

2.1 Previously Unpublished Variants

Two-Status-Bit Quotient Filter. Pandey et al. [16] already proposed a 2 status bit variant of their counting quotient filter. Their implementation however has average query and insertion times in $\Theta(n)$. The goal of our 2-status bit variant is to achieve running times close to $O(\textit{supercluster length})$. To achieve this, we change the definition of the fingerprint (only for this variant) such that no remainder can be zero $f' : x \mapsto \{0, ..., 2^q - 1\} \times \{1, ..., 2^r - 1\}$. Obtaining a non-zero remainder can easily be achieved by rehashing an element with different hash functions until the remainder is non-zero. This change to the fingerprint only has a minor impact on the false positive rate of the quotient filter ($n/m \cdot (2^r - 1)^{-1}$ instead of $n/m \cdot 2^{-r}$).

Given this change to the fingerprint we can easily distinguish empty slots from filled ones. Each slot uses two status bits the *occupied-bit* (first status bit in the description above) and the *new-run-bit* (run-delimiter). Using these status bits we can find a particular run by going to the left until we find a free slot and then counting the number of occupied- and new-run-bits while moving right from there (*Note*: a cluster start within a larger supercluster cannot be recognized when moving left).

Linear Probing Quotient Filter. This quotient filter is a hybrid between a linear probing hash table and a quotient filter. It uses no reordering of stored remainders and no status bits. To offset the removal of status bits we add three additional bits to the remainder (leading to a longer fingerprint). Similar to the two-status-bit quotient filter above we ensure no element x has $rem(x) = 0$ (by adapting the fingerprint function).

During the insertion of an element x the remainder $rem(x)$ is stored in the first empty slot after its canonical slot. Without status bits and reordering it is impossible to reconstruct the fingerprint of each inserted element. Therefore, a query looking for an element x compares its remainder $rem(x)$ with every remainder stored between x's canonical slot and the next empty slot. There are potentially more remainders that are compared than during the same operation on a normal quotient filter. The increased remainder length however reduces the chance of a single comparison to lead to a false positive by a factor of 8. Therefore, as long as the average number of compared remainders is less than 8 times more than before, the false positive remains the same or improves. In our tests, we found this to be true for fill degrees up to $\approx 70\%$. The number of compared remainders corresponds to the number of slots probed by a linear probing hash table, this gives the bound below. It should be noted that linear probing quotient filters cannot support deletions or the bounded growing technique.

▶ **Corollary 1** (p^+ linear probing q.f.). *The false positive rate of a linear probing quotient filter with $\delta = n/m$ (bound due to Knuth [10] chapter 6.4) is*

$$p^+ = \frac{E[\#comparisons]}{2^{r+3}-1} = \frac{1}{2}\left(1 + \frac{1}{(1-\delta)^2}\right) \cdot \frac{1}{2^{r+3}-1}$$

3 Concurrent Quotient Filter

Why does our concurrent quotient filter (not the Concurrent Linear Probing filter variant) still use locking? There are *Some Problems Inherent* to lock-free quotient filters. Every query has to read multiple data entries in a consistent state to succeed. All commonly known variants (except the linear probing quotient filter) use at least two status bits per slot, i.e., the *occupied-bit* and some kind of *run-delimiter-bit* (the run-delimiter might take different forms). The remainders and the run-delimiter-bit that belong to one slot cannot reliably be stored close to their slot and its occupied-bit. Therefore, the occupied-bit and the run-delimiter-bit cannot be updated with one atomic operation. For this reason, implementing a lock-free quotient filter that uses status bits would cause significant overheads (i.e. additional memory or multiple passes over the accessed data). The problem is that reading parts of multiple different but individually consistent states of the quotient filter leads to an inconsistent perceived state. To show this we assume that insertions happen atomically and transfer the table from one consistent state directly into the new final consistent state. During a query we have to find the canonical run and scan the remainders within this run. To find the canonical run we have to compute its rank among non-empty runs using the occupied-bits (either locally by iterating over slots or globally using a rank-select-data structure) and then find the run with the same rank using the run-delimiter-bits. There is no way to guarantee that the overall state has not changed between finding the rank of the occupied-bit bit and finding the appropriate run-delimiter. Specifically we might find a new run that was created by an insertion that shifted the actual canonical run. Similar things can happen when accessing the remainders especially when remainders are not stored interleaved with their corresponding status bits. Some of these issues can be fixed using multiple passes over the data but ABA problems might arise in particular when deletions are possible. To avoid these problems while

maintaining performance we have two options: either comparing remainders from different canonical slots and removing the need for status bits (i.e. the linear probing quotient filter) or by using locks that protect the relevant range of the table.

Besides the correctness there are two main goals for any concurrent data structure: scalability and overall performance. One major performance advantage of quotient filters over other AMQ data structures is their cache efficiency. Concurrent quotient filters should attempt to preserve this advantage. Especially insertions into short or empty clusters and (unsuccessful) queries with empty canonical runs lead to operations with only one or two accessed cache lines and at most one write operation. Any variant using external locks has an immediate disadvantage here.

Concurrent Slot Storage. Whenever data is accessed by multiple threads concurrently, we have to think about the atomicity of operations. The slots of a quotient filter can have arbitrary sizes. To reduce the number of updated cache lines, and to store all data connected to one slot together in one atomic data element we alternate between status bits and remainders. We have to take into account the memory that is wasted by using atomic data types inefficiently. It would be common to just use the smallest data type that can hold both the remainder and status bits and waste any excess memory. But quotient filters are designed to be space efficient. Therefore, we use the largest common atomic data type (64 bit) and pack as many slots as possible into one data element – we call this packed construct a group (of slots). This way, the waste of multiple slots accumulates to encompass additional elements. Furthermore, using this technique we can atomically read or write multiple slots at once – allowing us to update in bulk and even avoid some locking.

3.1 Concurrent Linear Probing Quotient Filter

Operations on a linear probing quotient filter (as described in Section 2.1) can be executed concurrently similar to operations of a linear probing hash table [11]. The table is constructed out of grouped atomic slots as described above. Each insertion changes the table atomically using a single compare and swap instruction. This implementation is even lock free, because at least one competing write operation on any given cell is successful. Queries find all remainders that were inserted into their canonical slot, because they are stored between their canonical slot and the next empty slot and the contents of a slot never change once something is stored within it.

3.2 Concurrent Quotient Filter with Local Locking

In this section we introduce an easy way to implement concurrent quotient filters without any memory overhead and without increasing the number of cache lines that are accessed during operations. This concurrent implementation is based on the basic (3-status-bit) quotient filter. The same ideas can also be implemented with the 2-status-bit variant, but there are some problems discussed later in this section.

Using status bits for local locking. To implement our locking scheme we use Two Combinations of Status Bits that are impossible to occur naturally (see Table 1) – `010` and `110`. We use `110` to implement a write lock. It is written into the first free slot after the canonical supercluster at the beginning of each write operation (using a CAS-operation see Block B Algorithm 1.a). Insertions wait when encountering a write lock. This ensures that only one insert operation can be active per supercluster. The combination `010` is used as a read lock.

Algorithm 1 Concurrent Operations.

1.a Insertion

```
(quot, rem) ← f(key)
// Block A: try a trivial insertion
group ← atomically load data around quot
if insertion into group is trivial then
    finish insertion with a CAS and return
// Block B: write lock the supercluster
scan right from it ← quot
    if it is write locked then
        wait until released and continue
    if it is empty then
        lock it with write lock and break
            if CAS unsuccessful re-examine it
// Block C: read lock the cluster
scan left from it ← quot
    if it is read locked then
        wait until released and retry this slot
    if it is cluster start then
        lock it with read lock and break
            if CAS unsuccessful re-examine it
// Block D: find the correct run
occ = 0;   run = 0
scan right from it
    if it is occupied and it < quot then occ++
    if it is run start then run++
    if occ = run and it ≥ quot then break
// Block E: insert into the run and shift
scan right from it
    if it is read locked then
        wait until released
    store rem in correct slot
    shift content of following slots
    (keep groups consistent)
    break after overwriting the write lock
unlock the read lock
```

1.b Query

```
(quot, rem) ← f(key)
// Block F: try trivial query
group ← atomically load data around slot quot
if answer can be determined from group then
    return this answer
// Block G: read lock the cluster
scan left from it ← quot
    if it is read locked then
        wait until released and retry this slot
    if it is cluster start then
        lock it with read lock and break
            if CAS unsuccessful re-examine it
// Block H: find the correct run
occ = 0;   run = 0
scan right from it
    if it is occupied and it < quot then occ++
    if it is run start then run++
    if occ = run and it ≥ quot then break
// Block I: search remainder within the run
scan right from it
    if it = rem
        unlock the read lock
        return contained
    if it is not continuation of this run
        unlock the read lock
        return not contained
```

All operations (both insertions Block C Algorithm 1.a and queries Block G Algorithm 1.b) replace the status bits of the first element of their canonical cluster with `010`. Additionally, inserting threads wait for each encountered read lock while moving elements in the table, see Block E Algorithm 1.a. When moving elements during the insertion each encountered cluster start is shifted and becomes part of the canonical cluster that is protected by the insertion's original read lock. This ensures, that no operation can change the contents of a cluster while it is protected with a read lock. It would also be possible to implement deletions in a similar way to insertions (first acquiring a write lock, then a read lock). But both queries and insertions are impacted by the possibility of holes being created within clusters while acquiring a lock requiring some extra work. The data structure used in our tests does not have deletions enabled.

Avoiding locks. Many instances of locking can be avoided, e.g., when the canonical slot for an insertion is empty (write the elements with a single compare and swap operation), when the canonical slot of a query either has no run (first status bit), or stores the wanted fingerprint. In addition to these trivial instances of lock elision, where the whole operation happens in one slot, we can also profit from our grouped atomic storage scheme. Since we store multiple slots together in one atomic data member, multiple slots can be changed at once. Each operation can act without acquiring a lock, if the whole operation can be completed without loading another data element. The correctness of the algorithm is still guaranteed, because the slots within one data element are always in a consistent state.

2-Status-Bit Concurrent Quotient Filter. In the 2-status-bit variant of the quotient filter there are no unused status bit combinations that can be used as read or write locks. But zero can never be a remainder when using this variant, therefore, a slot with an empty remainder but non-zero status bits cannot occur. We can use such a cell to represent a lock. Using this method, cluster starts within a larger supercluster cannot be recognized when scanning to the left ,i.e., in Blocks C and G (of Algorithms 1.a and 1.b). Instead we can only recognize supercluster starts (a non-empty cell to the right of an empty cell).

To write lock a supercluster, we store `01` in the status bits of an otherwise empty slot after the supercluster. To read lock a supercluster we remove the remainder from the table and store it locally until the lock is released (status bits `11`). However, this concurrent variant is not practical because we have to ensure that the read-locked slot is still a supercluster start (the slot to its left remains empty). To do this we can atomically compare and swap both slots. However this is a problem since both slots might be stored in different atomic data types or even cache lines. The variant is still interesting from a theoretical perspective where a compare and swap operation changing two neighboring slots is completely reasonable (usually below 64 bits). However, we have implemented this variant when we did some preliminary testing with non-aligned compare and swap operations and it is non-competitive with the other implementations.

Growing concurrently. The bounded growing technique (described in Section 2) can be used to increase the capacity of a concurrent quotient filter similar to that of a sequential quotient filter. In the concurrent setting we have to consider two things: distributing the work of the migration between threads and ensuring that no new elements are inserted into parts of the old table that were already migrated (otherwise they might be lost).

To distribute the work of migrating elements, we use methods similar to those in Maier et al. [11]. After the migration is triggered, every thread that starts an operation first helps with the migration before executing the operation on the new table. Reducing interactions between threads during the migration is important for performance. Therefore, we migrate the table in blocks. Every thread acquires a block by incrementing a shared atomic variable. The migration of each block happens one supercluster at a time. Each thread migrates all superclusters that begin in its block. This means that a thread does not migrate the first supercluster in its block, if it starts in the previous block. It also means that the thread migrates elements from the next block, if its last supercluster crosses the border to that block. The order of elements does not change during the migration, because they remain ordered by their fingerprint. In general this means that most elements within one block of the original table are moved into one of two blocks in the target table (block i is moved to $2i$ and $2i + 1$). By assigning clusters depending on the starting slot of their supercluster, we enforce that there are no two threads accessing the same slot of the target table. Hence, no atomic operations or locks are necessary in the target table.

As described before, we have to ensure that ongoing insert operations either finish correctly or help with the migration before inserting into the new table. Ongoing queries also have to finish to prevent deadlocks. To prevent other threads from inserting elements during the migration, we write lock each empty slot and each supercluster before it is migrated. These *migration-write-locks* are never released. To differentiate migration-write-locks from the ones used in a normal insertions, we write lock a slot and store a non-zero remainder (write locks are usually only stored in empty slots). This way, an ongoing insertion recognizes that the encountered write lock belongs to a migration. The inserting thread first helps with the migration before restarting the insertion after the table is fully migrated. Queries can happen concurrently with the migration, because the migration does not need read locks.

4 Fully Expandable QFs

The goal of this fully expandable quotient filter is to offer a resizable quotient filter variant with a bounded false positive rate that works well even if there is no known bound to the number of elements inserted. Adding new fingerprint bits to existing entries is impossible without access to the inserted elements. We adapt a technique that was originally introduced for scalable Bloom filters [2]. Once a quotient filter is filled, we allocate a new *additional* quotient filter. Each subsequent quotient filter increases the fingerprint size. Overall, this ensures a bounded false positive rate. This old idea offers new and interesting possibilities when applied to concurrent quotient filters, i.e., avoiding locks on lower levels, growing each level using the bounded growing technique, higher fill degree through cascading inserts, and early rejection of queries also through cascading inserts.

This new data structure starts out with one quotient filter, but over time, it may contain a set of quotient filters we call levels. At any point in time, only the newest (highest) level is active. Insertions operate on the active level. The data structure is initialized with two user-defined parameters the *expected capacity* c and the upper *bound for the false positive rate* $\overline{p}^+$. The first level table is initialized with m_0 slots where $m_0 = 2^{q_0}$ is the first power of 2 where $\delta_{grow} \cdot m_0$ is larger than c, here δ_{grow} is the fill ratio where growing is triggered and $\overline{n_i} = \delta_{grow} \cdot m_i$ is the maximum number of elements level i can hold. The number of remainder bits r_0 is chosen such that $\overline{p}^+ > 2\delta_{grow} \cdot 2^{-r_0}$ ($k_0 = q_0 + r_0$ fingerprint bits).

Queries have to check each level. Within the lower levels queries do not need any locks, because the elements there are finalized. The query performance depends on the number of levels. To keep the number of levels small, we have to increase the capacity of each subsequent level. To also bound the false positive rate, we have to reduce the false positive rate of each subsequent level. To achieve both of these goals, we increase the size of the fingerprint k_i by two for each subsequent level ($k_i = 2 + k_{i-1}$). Using the longer fingerprint we can ensure that once the new table holds twice as many elements as the old one ($\overline{n_i} = \overline{n_{i+1}}/2$), it still has half the false positive rate ($\overline{p}_i^+ = \overline{n_i} \cdot 2^{-k_i} = 2\overline{p}_{i+1}^+ = 2 \cdot \overline{n_{i+1}} \cdot 2^{-k_{i+1}}$).

When one level reaches its maximum capacity $\overline{n_i}$ we allocate a new level. Instead of allocating the new level to immediately have twice the number of slots as the old level, we allocate it with one 8th of the final size, and use the bounded growing algorithm (described in Section 2) to grow it to its final size (three growing steps). This way, the table has a higher average fill rate (at least $2/3 \cdot \delta_{grow}$ instead of $1/3 \cdot \delta_{grow}$).

▶ **Theorem 2** (Bounded p^+ in expandable QF)**.** *The fully expandable quotient filter holds the false positive probability $\overline{p}^+$ set by the user independently of the number of inserted elements.*

Proof. For the following analysis, we assume that fingerprints can potentially have an arbitrary length. The analysis of the overall false positive rate p^+ is very similar to that of the scalable Bloom filter. A false positive occurs if one of the ℓ levels has a false positive $p^+ = 1 - \prod_i (1 - p_i^+)$. This can be approximated with the Weierstrass inequality $p^+ \leq \sum_{i=1}^{\ell} p_i^+$. When we insert the shrinking false positive rates per level ($p_{i+1}^+ = p_i^+/2$) we obtain a geometric sum which is bounded by $2p_1^+$: $\sum_{i=0}^{\ell-1} p_i^+ 2^{-i} \leq 2p_1^+ < \overline{p}^+$. ◀

Using this growing scheme the number of filters is in $O(\log n/T)$, therefore, the bounds for queries are similar to those in a broad tree data structure. But due to the necessary pointers tree data structures take significantly more memory. Additionally, they are difficult to implement concurrently without creating contention on the root node.

Cascading Inserts. Cascading Inserts can be used to improve the memory usage and the query performance of our growing quotient filters. The idea is to insert elements on the lowest possible level. If the canonical slot in a lower level is empty, we insert the element into that level. This can be done using a simple CAS-operation (without acquiring a write lock). Queries on lower levels can still proceed without locking, because insertions cannot move existing elements.

The main reason to grow the table before it is full is to improve the performance by shortening clusters. The trade-off for this is space utilization. For the space utilization it would be optimal to fill each table 100%. Using cascading inserts this can be achieved while still having a good performance on each level. Queries on the lower level tables have no significant slow down due to cascading inserts, because the average cluster length remains small (cascading inserts lead to one-element clusters). Additionally, if we use cascading inserts, we can abort queries that encounter an empty canonical slot in one of the lower level tables, because this slot would have been filled by an insertion. Cascading inserts cause some overhead. Each insertion checks every level whether its canonical slot is empty. However, in some applications checking all levels is already necessary. For example in applications like element unification all lower levels are checked to prevent repeated insertions of one element. Applications like this often use combined query and insert operations that only insert if the element was not yet in the table. Here cascading inserts do not cost any overhead.

5 Experiments

All experiments were executed on a four socket Intel Xeon Gold 6138 machine with 20 cores per socket, each running at 2.0 GHz with 27.5MB cache size, and 768 GB main memory. The tests were compiled using gcc 9.2.0 and the operating system is Ubuntu 18.04.3. Each test was repeated 9 times using 3 different sequences of keys (same distribution) – 3 runs per sequence.

The first experiment we conducted is a speedup benchmark with strong work scaling between 1 to 160 threads (with hyperthreading) that can be seen in our technical report [12]. We compare our locally locked quotient filter (using status bit locking) and the lock-free linear probing filter with other AMQ data structures like a standard (externally) locked quotient filter, the counting quotient filter by Pandey et al. [14,16], and a specifically optimized bloom filter. The bloom filter was optimized by adapting its size to the size of the other tested data structures and in turn reducing the number of hash functions k such that a similar false positive rate was achieved ($k = 4$). We feel that this is the fairest comparison to the presented data structures (same memory; similar p^+). The results are fairly expected (linear scaling performance).

Fill Ratio Benchmark. In this benchmark, we test the same 5 filter implementations under a varying fill degree. To do this, we fill a table with $2^{30} \approx 1.1$G slots and $r = 10$. Every 10% of the fill ratio we execute a performance test the results are shown in Figure 2. During this test each table operation is executed 1M times. As one would expect, the throughput of our quotient filter variants decreases with increasing fill ratio. However, this seems not to be the case for the counting quotient filter and the Bloom filter. The optimized bloom filter that we used in these experiments is especially good at negative queries, because its density of 1s is actually very small, therefore, queries can often be aborted after one memory access. The insert performance of the counting quotient filter is at below 0.2MOps (independent of the fill degree). Our advanced quotient filter implementations – the linear probing quotient filter and

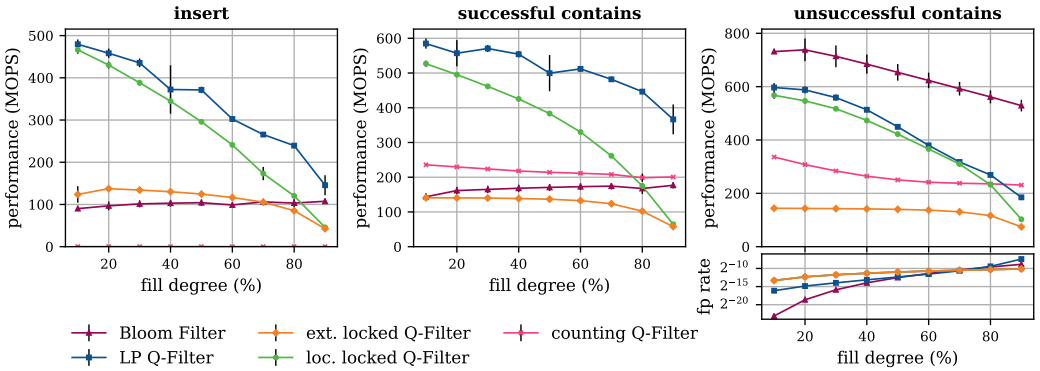

Figure 2 Throughput over fill degree. Using $p = 80$ threads and $2^{30} \approx 1.1$G slots. Each point of the measurement was tested using 1M operations (in parallel).

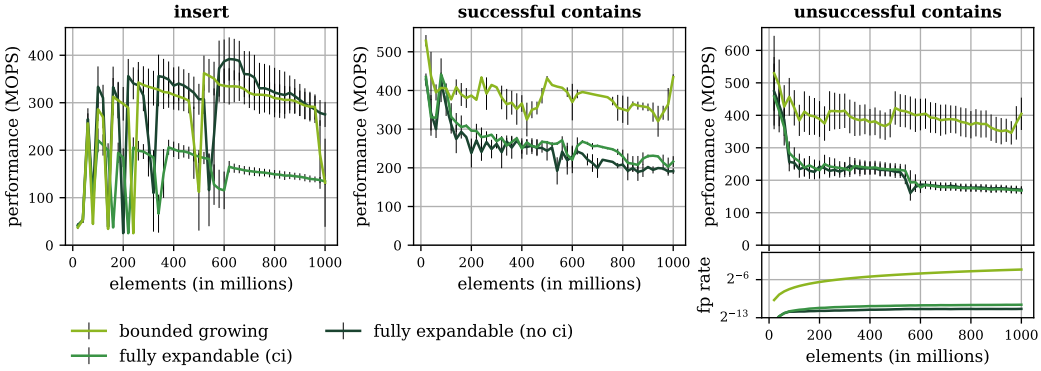

Figure 3 Throughput of the growing tables (all variants based on the local locking quotient filter). Using $p = 80$ threads 50 segments of 20M elements are inserted into a table initialized with $2^{24} \approx 16.8$M slots.

the locally locked quotient filter – display their strengths on sparser tables with about 3.25× and 2.89× higher insertion throughputs than the similar externally locked implementation at 30% fill degree. At 70% fill degree the advanced implementation are still 2.51× and 1.64× faster than the externally locked implementation.

Growing Benchmark. For this benchmark, shown in Figure 3, we insert 1G elements into each of our dynamically sized quotient filters bounded growing, fully expandable without cascading inserts (no ci), and fully expandable with cascading inserts (ci) (all based on the quotient filter using local locking). The filter was initialized with a capacity of only $2^{24} \approx 16.8$M (64× growing factor) slots and a target false positive rate of $\overline{p}^{+} = 2^{-10}$. The inserted elements are split into 50 segments of 20M elements each. We measure the running time of inserting each segment as well as the query performance after each segment.

As expected, the false positive rate of the quotient filter that uses bounded growing grows linearly with the number of elements, but each query still consists of only one table lookup – resulting in a query performance similar to that of the non-growing benchmark. Both fully expandable variants that combine bounded growing with the multi level technique stay below 2^{-10} false positive rate. But their query performance suffers due to the lower

level look ups and the additional memory accesses they incur. Cascading inserts can improve successful query times by over 10% (averaged over all segments), however, for unsuccessful queries we do not see an average speedup. As expected cascading inserts are slowing down insertion times significantly. AMQ-filter data structures are usually queried significantly more often than they are updated, therefore, many workloads in practice can still profit from this technique.

6 Conclusion

In this publication, we have shown a new technique for concurrent quotient filters that uses the status bits inherent to quotient filters for localized locking. Using this technique, no additional cache lines are accessed (compared to a sequential quotient filter). We were able to achieve a 1.6 times increase in insert performance over the external locking scheme ($p = 80$ and 70% fill degree; 2.1 and 2.4 on queries). Additionally, we proposed a simple linear probing based filter that does not use any status bits and is lock-free. Using the same amount of memory this filter achieves even better false positive rates up to a fill degree of 70% and also 1.5 times higher insertion speedups (than the local locking variant also at 70% fill degree).

We also propose to use the bounded growing technique which is supported by quotient filters to refine the growing scheme used in scalable Bloom filters. Using this combination of techniques guarantees that the overall data structure is always at least $2/3 \cdot \delta_{grow}$ filled (where δ_{grow} is the fill degree where growing is triggered). Using cascading inserts this can be improved by filling lower level tables even further, while also improving successful query times by up to 10%.

Our tests show that there is no optimal AMQ data structure. Which data structure performs best depends on the use case and the expected workload. The linear probing quotient filter is very good, if the table is not densely filled. The locally locked quotient filter is also efficient on tables below a fill degree of 70%. But, it is also more flexible for example when the table starts out empty and is filled to above 70% (i.e., constructing the filter). Our growing implementations work well if the number of inserted elements is not known prior to the table's construction. The counting quotient filter could perform well on very query heavy workloads that operate on densely filled tables.

There is some future work that could further improve the presented data structures. Two obvious improvements would be the implementation of deletions or the possibility to store element counters (similar to [16]). Additionally, it would be interesting to use hardware transactional memory to reduce the number of necessary locks even further. This has the possibility to speed up queries in particular.

References

1 Mohammad Al-hisnawi and Mahmood Ahmadi. Deep Packet Inspection using Quotient Filter. *IEEE Communications Letters*, 20(11):2217–2220, November 2016. doi:10.1109/LCOMM.2016.2601898.

2 Paulo Sérgio Almeida, Carlos Baquero, Nuno Preguiça, and David Hutchison. Scalable Bloom Filters. *Inf. Process. Lett.*, 101(6):255–261, March 2007. doi:10.1016/j.ipl.2006.10.007.

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. *Proc. VLDB Endow.*, 5(11):1627–1637, July 2012. doi:10.14778/2350229.2350275.

4 Alex D. Breslow and Nuwan S. Jayasena. Morton filters: fast, compressed sparse cuckoo filters. *The VLDB Journal*, August 2019. doi:10.1007/s00778-019-00561-0.

5 Pedro Celis, Per-Ake Larson, and J Ian Munro. Robin Hood Hashing. In *26th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 281–288. IEEE, 1985.

6 J. G. Cleary. Compact Hash Tables Using Bidirectional Linear Probing. *IEEE Transactions on Computers*, C-33(9):828–834, 1984.

7 Yan Collet. xxHash. `https://github.com/Cyan4973/xxHash`. Accessed March 21, 2019.

8 Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. Cuckoo Filter: Practically better than Bloom. In *10th ACM International on Conference on Emerging Networking Experiments and Technologies*, pages 75–88, 2014. `doi:10.1145/2674005.2674994`.

9 Afton Geil, Martin Farach-Colton, and John D. Owens. Quotient Filters: Approximate membership queries on the GPU. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 451–462, May 2018. `doi:10.1109/IPDPS.2018.00055`.

10 Donald Ervin Knuth. *The Art of Computer Programming: sorting and searching*, volume 3. Pearson Education, 1997.

11 Tobias Maier, Peter Sanders, and Roman Dementiev. Concurrent Hash Tables: Fast and general(?)! *ACM Trans. Parallel Comput.*, 5(4):16:1–16:32, February 2019. `doi:10.1145/3309206`.

12 Tobias Maier, Peter Sanders, and Robert Williger. Concurrent Expandable AMQs on the basis of quotient filters. *CoRR*, 2019. `arXiv:1911.08374`.

13 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. `doi:10.1137/1.9781611975055.4`.

14 Prashant Pandey. Counting Quotient Filter. `https://github.com/splatlab/cqf`. Accessed August 07, 2019.

15 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. *Cell Systems*, 7(2):201–207.e4, 2018. `doi:10.1016/j.cels.2018.05.021`.

16 Prashant Pandey, Michael A. Bender, Rob Johnson, and Rob Patro. A General-Purpose Counting Filter: Making every bit count. In *ACM Conference on Management of Data (SIGMOD)*, pages 775–787, 2017. `doi:10.1145/3035918.3035963`.