



WIKIPEDIA
The Free Encyclopedia

[Main page](#)

[Contents](#)

[Featured content](#)

[Current events](#)

[Random article](#)

[Donate to Wikipedia](#)

[Wikipedia store](#)

Interaction

[Help](#)

[About Wikipedia](#)

[Community portal](#)

[Recent changes](#)

[Contact page](#)

Tools

[What links here](#)

[Related changes](#)

[Upload file](#)

[Special pages](#)

[Permanent link](#)

[Page information](#)

[Wikidata item](#)

[Cite this page](#)

Print/export

[Create a book](#)

[Download as PDF](#)

[Printable version](#)

Languages

[Add links](#)

Article [Talk](#)

[Read](#) [Edit](#) [View history](#)

Search

Quotient filter

From Wikipedia, the free encyclopedia

A **quotient filter**, introduced by Bender *et al.* in 2011,^{[1][2]} is a space-efficient [probabilistic data structure](#) used to test whether an [element](#) is a member of a [set](#) (an [approximate member query](#) filter, AMQ). A query will elicit a reply specifying either that the element is definitely not in the set or that the element is probably in the set. The former result is definitive; *i.e.*, the test does not generate [false negatives](#). But with the latter result there is some probability, ϵ , of the test returning "element is in the set" when in fact the element is not present in the set (*i.e.*, a [false positive](#)). There is a tradeoff between ϵ , the false positive rate, and storage size; increasing the filter's storage size reduces ϵ . Other AMQ operations include "insert" and "optionally delete". The more elements are added to the set, the larger the probability of false positives.

A typical application for quotient filters, and other AMQ filters, is to serve as a proxy for the keys in a [database](#) on disk. As keys are added to or removed from the database, the filter is updated to reflect this. Any lookup will first consult the fast quotient filter, then look in the (presumably much slower) database only if the quotient filter reported the presence of the key. If the filter returns absence, the key is known not to be in the database without any disk accesses having been performed.

A quotient filter has the usual AMQ operations of insert and query. In addition it can also be merged and re-sized without having to re-hash the original keys (thereby avoiding the need to access those keys from secondary storage). This property benefits certain kinds of [log-structured merge-trees](#).

Contents [hide]

1 Algorithm description

1.1 Lookup

1.2 Lookup example

1.3 Insertion

1.4 Insertion example

2 Cost/performance

2.1 Cluster length

2.2 Probability of false positives

2.3 Space/performance

3 Application

4 See also

5 Notes

Part of a series on

Probabilistic data structures

[Bloom filter](#) · [Count–min sketch](#) ·

[Quotient filter](#) · [Skip list](#)

Random trees

[Random binary tree](#) · [Treap](#) ·

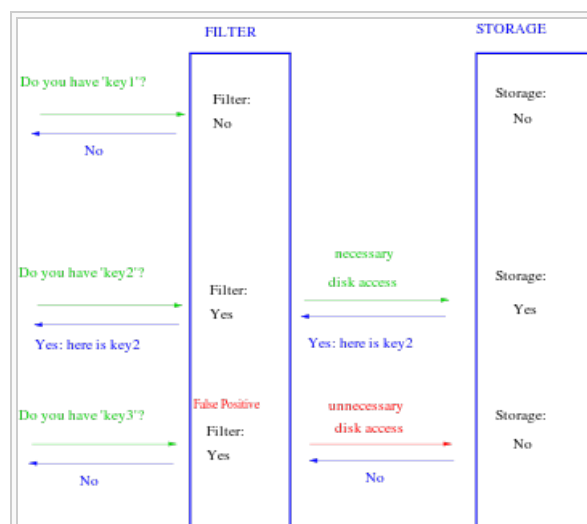
[Rapidly exploring random tree](#)

Related

[Randomized algorithm](#)

[Computer science portal](#)

v · t · e



An approximate member query (AMQ) filter used to speed up answers in a key-value storage system. Key-value pairs are stored on a disk which has slow access times. AMQ filter decisions are much faster. However some unnecessary disk accesses are made when the filter reports a positive (in order to weed out the false positives). Overall answer speed is better with the AMQ filter than without it. Use of an AMQ filter for this purpose, however, does increase memory usage.

Algorithm description [edit]

The quotient filter is a *compact* [hash table](#). Cleary^[3] defines a compact hash table as one in which the table entries contain only a portion of the key plus some additional meta-data bits. These bits are used to deal with the case when distinct keys happen to hash to the same table entry. By way of contrast, other types of hash

tables that deal with such collisions by linking to overflow areas are not compact because the overhead due to linkage can exceed the storage used to store the key.^[3] In a quotient filter a [hash function](#) generates a p -bit fingerprint. The r least significant bits is called the remainder while the $q = p - r$ most significant bits is called the quotient, hence the name *quotienting* (coined by Knuth.^[4]) The hash table has 2^q slots.

For some key d which hashes to the fingerprint d_H , let its quotient be d_Q and the remainder be d_R . QF will try to store the remainder in slot d_Q , which is known as the *canonical slot*. However the canonical slot might already be occupied because multiple keys can hash to the same fingerprint—a *hard collision*—or because even when the keys' fingerprints are distinct they can have the same quotient—a *soft collision*. If the canonical slot is occupied then the remainder is stored in some slot to the right.

As described below, the insertion algorithm ensures that all fingerprints having the same quotient are stored in contiguous slots. Such a set of fingerprints is defined as a *run*.^[1] Note that a run's first fingerprint might not occupy its canonical slot if the run has been forced right by some run to the left.

However a run whose first fingerprint occupies its canonical slot indicates the start of a *cluster*.^[1] The initial run and all subsequent runs comprise the cluster, which terminates at an unoccupied slot or the start of another cluster.

The three additional bits are used to reconstruct a slot's fingerprint. They have the following function:

- **is_occupied** is set when a slot is the canonical slot for some key stored (somewhere) in the filter (but not necessarily in this slot).
- **is_continuation** is set when a slot is occupied but not by the first remainder in a run.
- **is_shifted** is set when the remainder in a slot is not in its canonical slot.

The various combinations have the following meaning:

```
is_occupied
  is_continuation
    is_shifted
0 0 0 : Empty Slot
0 0 1 : Slot is holding start of run that has been shifted from its canonical slot.
0 1 0 : not used.
0 1 1 : Slot is holding continuation of run that has been shifted from its canonical
slot.
1 0 0 : Slot is holding start of run that is in its canonical slot.
1 0 1 : Slot is holding start of run that has been shifted from its canonical slot.
        Also the run for which this is the canonical slot exists but is shifted
right.
1 1 0 : not used.
1 1 1 : Slot is holding continuation of run that has been shifted from its canonical
slot.
        Also the run for which this is the canonical slot exists but is shifted
right.
```

Lookup [\[edit\]](#)

We can test if a quotient filter contains some key, d , as follows.^[1]

We hash the key to produce its fingerprint, d_H , which we then partition into its high-order q bits, d_Q , which comprise its quotient, and its low-order r bits, d_R , which comprise its remainder. Slot d_Q is the key's canonical slot. That slot is empty if its three meta-data bits are false. In that case the filter does not contain the key.

If the canonical slot is occupied then we must locate the quotient's run. The set of slots that hold remainders belonging to the same quotient are stored contiguously and these comprise the quotient's run. The first slot in the run might be the canonical slot but it is also possible that the entire run has been shifted to the right by the encroachment from the left of another run.

To locate the quotient's run we must first locate the start of the cluster. The cluster consists of a contiguous set of runs. Starting with the quotient's canonical slot we can scan left to locate the start of the cluster, then scan right to locate the quotient's run.

We scan left looking for a slot with *is_shifted* is false. This indicates the start of the cluster. Then we scan right keeping a running count of the number of runs we must skip over. Each slot to the left of the canonical slot having *is_occupied* **set** indicates another run to be skipped, so we increment the running count. Each slot having *is_continuation* **clear** indicates the start of another run, thus the end of the previous run, so we decrement the running count. When the running count reaches zero, we are scanning the quotient's run. We

can compare the remainder in each slot in the run with d_R . If found, we report that the key is (probably) in the filter otherwise we report that the key is definitely not in the filter.

Lookup example [\[edit\]](#)

Take, for example, looking up element e . See state 3 in the figure. We would compute $\text{hash}(e)$, partition it into its remainder, e_R and its quotient e_Q which is 4. Scanning left from slot 4 we encounter three *is_occupied* slots, at indexes 4, 2 and 1, indicating e_Q 's run is the 3rd run in the cluster. The scan stops at slot 1, which we detect as the cluster's start because it is not empty and not shifted. Now we must scan right to the 3rd run. The start of a

run is indicated by *is_continuation* being false. The 1st run is found at index 1, the 2nd at 4 and the 3rd at 5. We compare the remainder held in each slot in the run that starts at index 5. There is only one slot in that run but, in our example, its remainder equals e_R indicating that e is indeed a member of the filter, with probability $1 - \epsilon$.

Insertion [\[edit\]](#)

Insertion follows a path similar to lookup until we ascertain that the key is definitely not in the filter.^[1] At that point we insert the remainder in a slot in the current run, a slot chosen to keep the run in sorted order. We shift forward the remainders in any slots in the cluster at or after the chosen slot and update the slot bits.

- Shifting a slot's remainder does not affect the slot's *is_occupied* bit because it pertains to the slot, not the remainder contained in the slot.
- If we insert a remainder at the start of an existing run, the previous remainder is shifted and becomes a continuation slot, so we set its *is_continuation* bit.
- We set the *is_shifted* bit of any remainder that we shift.

Insertion example [\[edit\]](#)

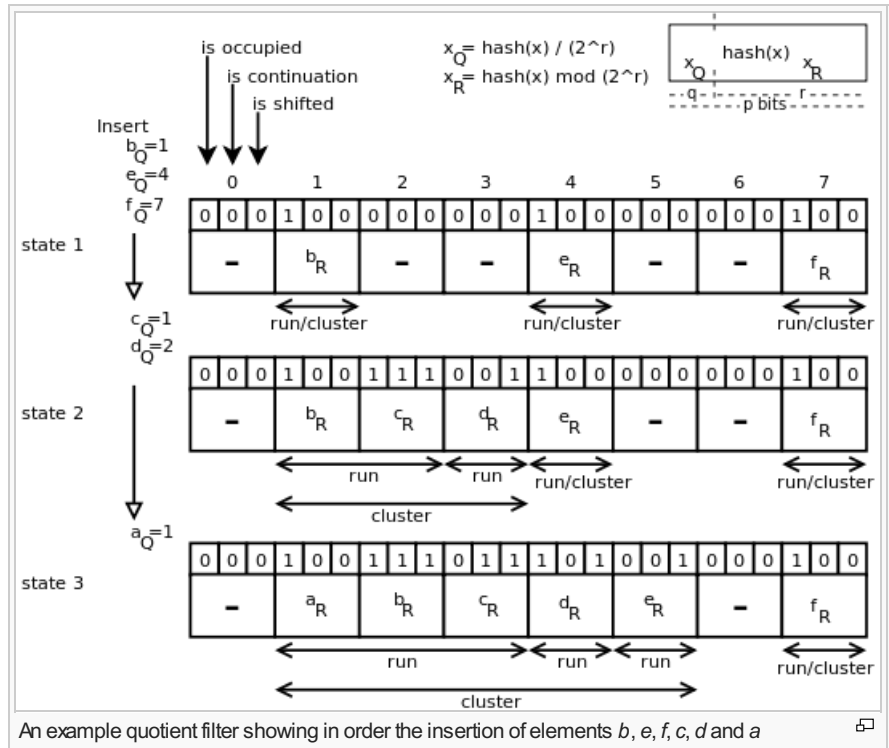
The figure shows a quotient filter proceeding through a series of states as elements are added. In state 1 three elements have been added. The slot each one occupies forms a one-slot run which is also a distinct cluster.

In state 2 elements c and d have been added. Element c has a quotient of 1, the same as b . We assume $b_R < c_R$ so c_R is shifted into slot 2, and is marked as both a *continuation* and *shifted*. Element d has a quotient of 2. Since its canonical slot is in use, it is shifted into slot 3, and is marked as *shifted*. In addition its canonical slot is marked as *occupied*. The runs for quotients 1 and 2 now comprise a cluster.

In state 3 element a has been added. Its quotient is 1. We assume $a_R < b_R$ so the remainders in slots 1 through 4 must be shifted. Slot 2 receives b_R and is marked as a *continuation* and *shifted*. Slot 5 receives e_R and is marked as *shifted*. The runs for quotients 1, 2 and 4 now comprise a cluster, and the presence of those three runs in the cluster is indicated by having slots 1, 2 and 4 being marked as *occupied*.

Cost/performance [\[edit\]](#)

Cluster length [\[edit\]](#)



Bender^[1] argues that clusters are small. This is important because lookups and inserts require locating the start and length of an entire cluster. If the hash function generates uniformly distributed fingerprints then the length of most runs is $O(1)$ and it is highly likely that *all* runs have length $O(\log m)$ where m is the number of slots in the table.^[1]

Probability of false positives [\[edit\]](#)

Bender^[1] calculates the probability of a false positive (i.e. when the hash of two keys results in the same fingerprint) in terms of the hash table's remainder size and load factor. Recall that a p bit fingerprint is partitioned into a q bit quotient, which determines the table size of $m = 2^q$ slots, and a r bit remainder. The load factor α is the proportion of occupied slots n to total slots m : $\alpha = n/m$. Then, for a good hash function, $1 - e^{-\alpha/2^r} \leq 2^{-r}$ is approximately the probability of a hard collision.

Space/performance [\[edit\]](#)

A quotient filter requires 10–25% more space than a comparable Bloom filter but is faster because each access requires evaluating only a single hash function.^[5]

Application [\[edit\]](#)

Quotient filters are AMQs and, as such, provide many of the same benefits as [Bloom filters](#). A large database, such as Webtable^[6] may be composed of smaller sub-tables each of which has an associated filter. Each query is distributed concurrently to all sub-tables. If a sub-table does not contain the requested element, its filter can quickly complete the request without incurring any I/O.

Quotient filters offer two benefits in some applications.

1. Two quotient filters can be efficiently merged without affecting their false positive rates. This is not possible with Bloom filters.
2. A few duplicates can be tolerated efficiently and can be deleted.

The space used by quotient filters is comparable to that of Bloom filters. However quotient filters can be efficiently merged within memory without having to re-insert the original keys.

This is particularly important in some log structured storage systems that use the [log-structured merge-tree](#) or LSM-tree.^[7] The LSM-tree is actually a collection of trees but which is treated as a single key-value store. One variation of the LSM-Tree is the [Sorted Array Merge Tree](#) or SAMT.^[5] In this variation, a SAMT's component trees are called [Wanna-B-trees](#). Each Wanna-B-tree has an associated quotient filter. A query on the SAMT is directed at only select Wanna-B-trees as evidenced by their quotient filters.

The storage system in its normal operation compacts the SAMT's Wanna-B-trees, merging smaller Wanna-B-trees into larger ones and merging their quotient filters. An essential property of quotient filters is that they can be efficiently merged without having to re-insert the original keys. Given that for large data sets the Wanna-B-trees may not be in memory, accessing them to retrieve the original keys would incur many I/Os.



By construction the values in a quotient filter are stored in sorted order. Each run is associated with a specific quotient value, which provides the most significant portion of the fingerprint, the runs are stored in order and each slot in the run provides the least significant portion of the fingerprint.






So, by working from left to right, one can reconstruct all the fingerprints and the resulting list of integers will be in sorted order. Merging two quotient filters is then a simple matter of converting each quotient filter into such a list, merging the two lists and using it to populate a new larger quotient filter. Similarly, we can halve or double the size of a quotient filter without rehashing the keys since the fingerprints can be recomputed using just the quotients and remainders.^[1]

See also [\[edit\]](#)

- [MinHash](#)
- [Bloom filter](#)

Notes [\[edit\]](#)

- ¹ ^a ^b ^c ^d ^e ^f ^g ^h ⁱ Bender, Michael A.; [Farach-Colton, Martin](#); Johnson, Rob; Kuzmaul, Bradley C.; Medjedovic, Dzejla; Montes, Pablo; Shetty, Pradeep; Spillane, Richard P.; Zadok, Erez (June 2011). "[Don't thrash: how to cache your hash on flash](#)"  (PDF). *Proceedings of the 3rd USENIX conference on Hot topics in storage and file systems (HotStorage'11)*. Retrieved 21 July 2012.
- ² ^a Bender, Michael A.; Farach-Colton, Martin; Johnson, Rob; Kraner, Russell; Kuzmaul, Bradley C.; Medjedovic, Dzejla; Montes, Pablo; Shetty, Pradeep; Spillane, Richard P.; Zadok, Erez (June 2011). "[Don't thrash: how to cache your hash on flash](#)"  (PDF). *Proceedings of the 3rd USENIX conference on Hot topics in storage and file systems (HotStorage'11)*. Retrieved 21 July 2012.

- Uzejla; Montes, Pablo; Shetty, Pradeep; Spillane, Richard P.; Zadok, Erez (2012). "Don't thrash: how to cache your hash on flash"  (PDF). *The Proceedings of the VLDB Endowment (PVLDB)* **5** (11): 1627–1637.
3. ^{[^](#)} ^{[a](#)} ^{[b](#)} Cleary, J.G. (1984). "Compact hash tables using bidirectional linear probing". *IEEE T. Comput.* **33** (9): 828–834. doi:[10.1109/TC.1984.1676499](https://doi.org/10.1109/TC.1984.1676499) .
 4. ^{[^](#)} Knuth, Donald (1973). *The Art of Computer Programming: Searching and Sorting*, volume 3. Section 6.4, exercise 13: Addison Wesley.
 5. ^{[^](#)} ^{[a](#)} ^{[b](#)} Spillane, Richard (May 2012). "Efficient, Scalable, and Versatile Application and System Transaction Management for Direct Storage Layers"  (PDF). Retrieved 21 July 2012.
 6. ^{[^](#)} Chang, Fay et al. (2006). "Bigtable: A Distributed Storage System for Structured Data"  (PDF). *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*: 15–15. Retrieved 21 July 2012.
 7. ^{[^](#)} O'Neil, Patrick et al. (1996). "The log-structured merge-tree (LSM-tree)". *Acta Informatica* **33** (4): 351–385. doi:[10.1007/s002360050048](https://doi.org/10.1007/s002360050048) .

Categories: [Hashing](#) | [Probabilistic data structures](#)

This page was last modified on 2 September 2015, at 01:52.

Text is available under the [Creative Commons Attribution-ShareAlike License](#); additional terms may apply. By using this site, you agree to the [Terms of Use](#) and [Privacy Policy](#). Wikipedia® is a registered trademark of the [Wikimedia Foundation, Inc.](#), a non-profit organization.

[Privacy policy](#) [About Wikipedia](#) [Disclaimers](#) [Contact Wikipedia](#) [Developers](#) [Mobile view](#)

