A General-Purpose Counting Filter: Making Every Bit Count

Prashant Pandey, Michael A. Bender, Rob Johnson, Rob Patro
Stony Brook University, NY
Approximate Membership Query (AMQ)

- An AMQ is a lossy representation of a set.
- Operations: inserts and membership queries.
- Compact space:
  - Often taking < 1 byte per item.
  - Comes at the cost of occasional false positives.
• A Bloom filter is a bit-array + $k$ hash functions.
(Here $k=2$.)
Insertions in a Bloom filter

- A Bloom filter is a bit-array + $k$ hash functions. (Here $k=2$.)
Insertions in a Bloom filter

- A Bloom filter is a bit-array + $k$ hash functions.
  (Here $k=2$.)
A Bloom filter is a bit-array + $k$ hash functions. (Here $k=2$.)
The Bloom filter has a bounded false-positive rate.

Membership query in a Bloom filter

query(W)

1 0 1 1 1 0 1

W X Y
Membership query in a Bloom filter

- The Bloom filter has a bounded false-positive rate.
Membership query in a Bloom filter

- The Bloom filter has a bounded false-positive rate.
Membership query in a Bloom filter

• The Bloom filter has a bounded false-positive rate.
Membership query in a Bloom filter

- The Bloom filter has a bounded false-positive rate.
Membership query in a Bloom filter

The Bloom filter has a bounded false-positive rate.
Bloom filters are ubiquitous

- Streaming applications
- Networking
- Databases
- Computational biology
- Storage systems
A counting filter is a lossy representation of a multiset.

- Operations: inserts, count, and delete.

- Generalizes AMQs
  - False positives \( \approx \) over-counts.
Why is counting important?

• Counting filters have numerous applications:
  • Computational biology, e.g., k-mer counting.
  • Network anomaly detection.
  • Natural language processing, e.g., n-gram counting.

• Counting enables AMQs to support deletes.
Many real data sets have skewed counts.

Counting filters should handle skewed data sets efficiently.
Many real data sets have skewed counts.

Counting filters should handle skewed data sets efficiently.
Counting Bloom filters

[Fan et al., 2000]

- Counters must be large enough to hold count of most frequent item.
- Counting Bloom filters are not space-efficient for skewed data sets.
Counting Bloom filters
[Fan et al., 2000]

- Counters must be large enough to hold count of most frequent item.
- Counting Bloom filters are not space-efficient for skewed data sets.

**RNA-seq dataset**
Total number of items: 19.6 Billion
Number of distinct items: 1.1 Billion
Maximum frequency: ~8 Million

Space usage of a CBF: ~38GB
This paper: The counting quotient filter (CQF)

- A replacement for the (counting) Bloom filter.
- Space and computationally efficient.
- Uses variable-sized counters to handle skewed data sets efficiently.

\[
\text{CQF space} \leq \text{BF space} + O(\sum_{x \in S} \log c(x))
\]

Asymptotically optimal
This paper: The counting quotient filter (CQF)

- A replacement for the (counting) Bloom filter.
- Space and computationally efficient.
- Uses variable-sized counters to handle skewed data sets efficiently.

RNA-seq dataset

Total number of items: 19.6 Billion
Number of distinct items: 1.1 Billion
Maximum frequency: ~8 Million

Space usage of a CQF: ~2.5GB
CQF space ≤ BF space + O(∑ x∈S log c(x))

Asymptotically optimal
Other features of the CQF

• Smaller than many non-counting AMQs
  • Bloom, cuckoo [Fan et al., 2014], and quotient [Bender et al., 2012] filters.

• Good cache locality

• Deletions

• Dynamically resizable

• Mergeable
Contributions

- New quotient filter metadata scheme
  - Smaller and faster than original quotient filter
- Efficient variable-length counter encoding method
  - Zero overhead for counters
- Fast implementation of bit-vector select on words
  - Exploits new x86 bit-manipulation instructions
Quotienting: An alternative to Bloom filters

- Store fingerprint compactly in a hash table.
  - Take a fingerprint $h(x)$ for each element $x$.

- Only source of false positives:
  - Two distinct elements $x$ and $y$, where $h(x) = h(y)$.
  - If $x$ is stored and $y$ isn’t, $\text{query}(y)$ gives a false positive.
Storing compact fingerprints

- \( b(x) \) = location in the hash table
- \( t(x) \) = tag stored in the hash table
Storing compact fingerprints

- $b(x) = \text{location in the hash table}$
- $t(x) = \text{tag stored in the hash table}$

Collisions in the hash table?
Storing compact fingerprints

- $b(x) = \text{location in the hash table}$
- $t(x) = \text{tag stored in the hash table}$

Collisions in the hash table? Linear probing.
Storing compact fingerprints

Does $t(v)$ belongs to bucket 4 or 5?

- The home bucket for $t(u)$ and $t(v)$ is 4.

Bucket index

Tag

bucket 4 or 5?
Resolving collisions in the CQF

- CQF uses two metadata bits to resolve collisions and identify the home bucket.

- The metadata bits group tags by their home bucket.
Resolving collisions in the CQF

- CQF uses two metadata bits to resolve collisions and identify the home bucket.

![Diagram showing the resolution of collisions with metadata bits]

- The metadata bits group tags by their home bucket.
Resolving collisions in the CQF

- CQF uses two metadata bits to resolve collisions and identify the home bucket.

- The metadata bits group tags by their home bucket.

The metadata bits enable us to identify the slots holding the contents of each bucket.
Counting quotient filter (CQF)

Implementation:
2 Meta-bits per slot.

\[ h(x) \rightarrow h_0(x) \parallel h_1(x) \]

Abstract Representation

\[ 0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \]
Counting quotient filter (CQF)

Implementation:
2 Meta-bits per slot.

\[ h(x) \rightarrow h_0(x) || h_1(x) \]

Abstract Representation

\[ 2^q \]

\[ 0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \]

\[ h(a) \]

runends

occupieds

1

1

\[ h_1(a) \]
Counting quotient filter (CQF)

Implementation:
2 Meta-bits per slot.

\[ h(x) \rightarrow h_0(x) \parallel h_1(x) \]

Abstract Representation

\[ 0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6 \ 7 \]

\[ h(a) \]

\[ h(b) \]

- \( \text{occupieds} \)
- \( \text{runends} \)
Counting quotient filter (CQF)

Implementation:
2 Meta-bits per slot.

\[ h(x) \rightarrow h_0(x) \ || \ h_1(x) \]

Abstract Representation
\[ \begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\end{array} \]

\[ \begin{array}{c}
\downarrow \\
h(a) \\
\downarrow \\
h(d) \\
\downarrow \\
h(b) \\
\end{array} \]

occupieds

runends
Counting quotient filter (CQF)

Implementation:
2 Meta-bits per slot.

\[ h(x) \rightarrow h_0(x) \, || \, h_1(x) \]

Abstract Representation
\[ 0 \, 1 \, 2 \, 3 \, 4 \, 5 \, 6 \, 7 \]

- \( h(a) \)
- \( h(b) \)
- \( h(d) \)
- \( h(e) \)

runends

occupieds

\[ h_1(a) \, h_1(b) \, h_1(d) \, h_1(e) \]
Counting quotient filter (CQF)

Implementation:
2 Meta-bits per slot.

$h(x) \rightarrow h_0(x) \| h_1(x)$
Counting quotient filter (CQF)

Implementation:
2 Meta-bits per slot.

\[ h(x) \rightarrow h_0(x) \parallel h_1(x) \]

Abstract Representation

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\
\hline
h(a) & h(d) & h(f) \\
h(b) & h(e) \\
h(c) \\
\end{array}
\]
Can accelerate metadata operations using x86 bit-manipulation instructions.

Asymptotic improvement in query performance over the original QF.

\[
\text{Rank}(\text{occupieds}, 3) = 2 \quad \text{Select}(\text{runends}, 2) = 5
\]
Encoding counts
Encoding counts

- Metadata scheme tells us the run of slots holding contents of a bucket.
- We can encode contents of buckets however we want.
- *The original quotient filter used repetition (unary).*
Encoding counts

- **We want to count in binary, not unary.**
- Idea: use some of the space for tags to store counts.
- Issue: determine which are tags and which are counts without using even one “control” bit.

![Diagram of encoding counts]

4 copies of $t(u)$
Dataset: 2 copies of 0, 7 copies of 3, and 9 copies 8.

- An encoding scheme to count the multiplicity of items.
- Variable-sized counter.
- Using slots reserved for remainders to, instead, store count information.
The CQF insert performance in RAM is similar to that of state-of-the-art non-counting AMQs.

The CQF is significantly faster at low load factors and slightly slower on high load factors.
Performance: Skewed datasets

- The CQF outperforms the CBF by a factor of 6x-10x on both inserts and lookups.

![Graphs showing performance comparison between CQF and CBF for inserts and lookups.](image)
Conclusion

• The CQF is smaller and faster than other AMQs, even ones that can’t count.

• The CQF also supports deletes, resizing, cache locality, and other features applications need.

• The CQF demonstrates the extensible design of the quotient filter.

https://github.com/splatlab/cqf
Space analysis: Bloom Filter

- \( m = \# \text{ of bits} \)
- \( n = \# \text{ of elements} \)
- \( k = \# \text{ of hash functions} \)
- \( k = \frac{m}{n \ln 2} \)
- \( S = \frac{m}{n} \)
- False-positive rate \( = 2^{-\frac{m}{n \ln 2}} = 2^{-S \ln 2} \)
Space analysis: Cuckoo Filter

- $f = \# \text{ of fingerprint bits}$
- $b = \# \text{ of entries in each bucket}$
- $\alpha = \text{load factor}$

- bits per element $S = \frac{\alpha}{f}$
- false-positive rate $= \frac{2b}{2^f} = \frac{2b}{2^{S\alpha}}$
Space analysis: Quotient filter

- \( q \) = # of quotient bits
- \( r \) = # of remainder bits
- \( c \) = # of metadata bits per slot
- \( \alpha \) = load factor
- # of slots = \( 2^q \)
- bits per element \( S = (r+c)/\alpha \)
- false-positive rate = \( \alpha 2^{-r} = \alpha 2^{-\alpha S+c} \)

The quotient filter always takes less space than the cuckoo filter and offers better false-positive rate than the Bloom filter whenever

\[
S \geq \frac{(c + \ln \alpha)}{(\alpha - \ln 2)}
\]