A General Purpose Counting Filter: Making Every Bit Count

Panday, et. al. 2018

Presented by Taher Mun

AMQ ->
cache-friendly AMQ ->
better cache-friendly AMQ ->
cache-friendly counting

Approximate Membership Query

- Is an element present in the set?
- If “NO”, then definitely not in the set
- If “YES”, then in set with high probability

Approximate Membership Query Structures

- Bloom filter
- Bloom filter extensions
- Cuckoo Filter
- Quotient Filter

Bloom filter

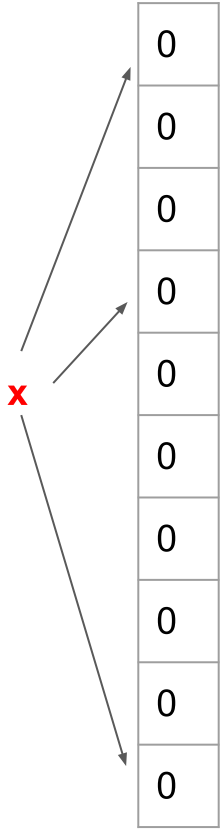
- Size *m* bit vector
- *k* hash functions
- “YES” if $h_i(x) = 1$ for $1 < i <= k$
- Else “NO”

0
0
0
0
0
0
0
0
0
0

Bloom filter

- Size *m* bit vector
- *k* hash functions
- "YES" if $h_i(x) = 1$ for $1 < i <= k$
- Else "NO"

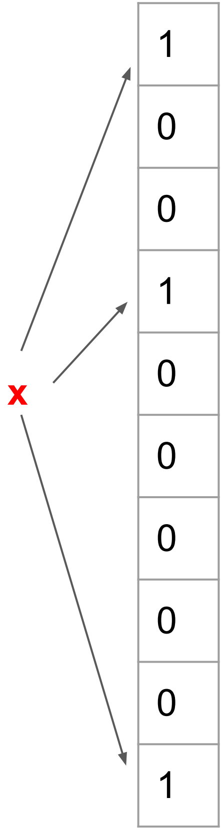
insert(x)



Bloom filter

- Size *m* bit vector
- *k* hash functions
- “YES” if $h_i(x) = 1$ for $1 < i <= k$
- Else “NO”

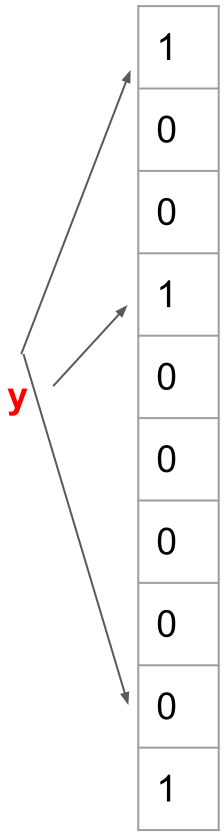
insert(x)



Bloom filter

- Size *m* bit vector
- *k* hash functions
- "YES" if $h_i(x) = 1$ for $1 < i <= k$
- Else "NO"

query(y)

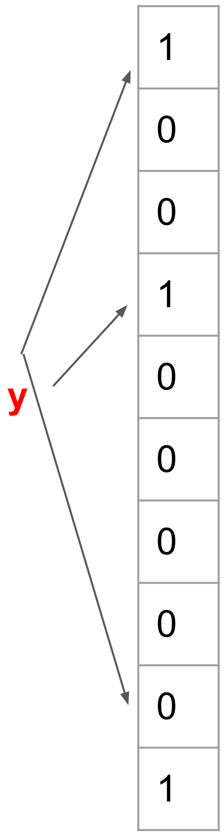


Bloom filter

- Size *m* bit vector
- *k* hash functions
- "YES" if $h_i(x) = 1$ for $1 < i <= k$
- Else "NO"

query(y)

X



Bloom Filter Drawbacks

- No delete support
- No resizing
- Bad locality
- No counting w/o extra bits

Quotient Filter (Bender, et al 2012)

- p-bit fingerprint divided into quotient *q* and remainder *r*
- *q* -> slot, *r* -> stored.
- run = contiguous slots [...i...] where each r_i has the same *q*
- 3 metadata bits for each slot:
  - Occupied - *q* of this slot associated with some *r* in table
  - Continuation - same *q* as prev slot
  - Shifted - current slot is not *q*
- Invariant: if $q < q'$, then *r* appears before *r'*

QF Query/Insert

- Linear probing
- “Shift” to right when needed

```
MAY-CONTAIN(A, f)
  f_q ← ⌊f/2^r⌋                ▷ quotient
  f_r ← f mod 2^r              ▷ remainder
  if ¬ is-occupied(A[f_q])
     then return FALSE
  ▷ walk back to find the beginning of the cluster
  b ← f_q
  while is-shifted(A[b])
        do DECR(b)
  ▷ walk forward to find the actual start of the run
  s ← b
  while b ≠ f_q
        do ▷ invariant: s points to first slot of bucket b
           ▷ skip all elements in the current run
           repeat INCR(s)
             until ¬ is-continuation(A[s])
           ▷ find the next occupied bucket
           repeat INCR(b)
             until is-occupied(A[b])
  ▷ s now points to the first remainder in bucket f_q
  ▷ search for f_r within the run
  repeat if A[s] = f_r
             then return TRUE
           INCR(s)
     until ¬ is-continuation(A[s])
  return FALSE
```

Figure 3: Algorithm for checking whether a fingerprint f is present in the QF A.

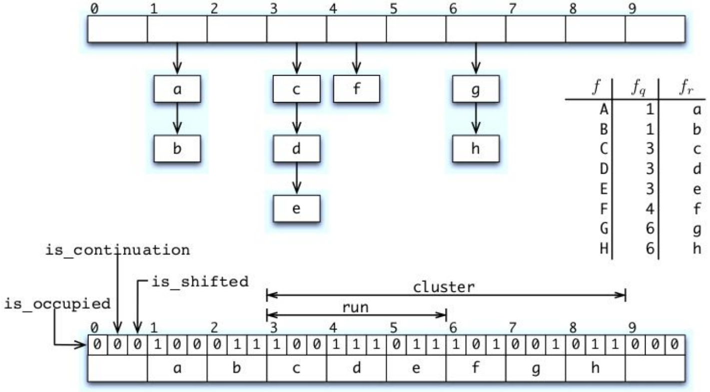
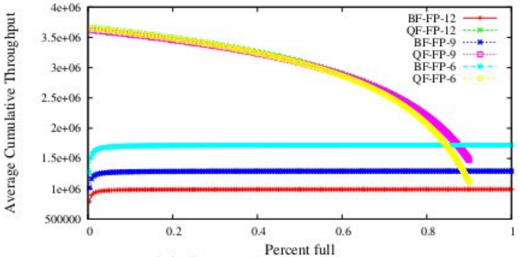


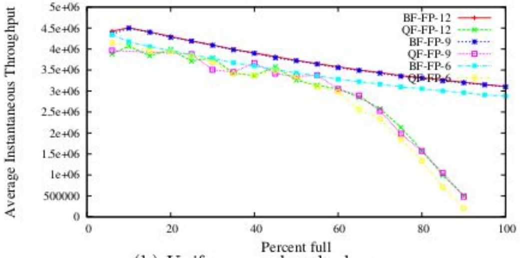
Figure 2: An example quotient filter with 10 slots along with its equivalent open hash table representation. The remainder, f_r, of a fingerprint f is stored in the bucket specified by its quotient, f_q. The quotient filter stores the contents of each bucket in contiguous slots, shifting elements as necessary and using three meta-data bits to enable decoding.

```
MAY-CONTAIN(A, f)
    f_q ← ⌊f/2^r⌋                          ▷ quotient
    f_r ← f mod 2^r                        ▷ remainder
    if ¬ is-occupied(A[f_q])
       then return FALSE
    ▷ walk back to find the beginning of the cluster
    b ← f_q
    while is-shifted(A[b])
          do DECR(b)
    ▷ walk forward to find the actual start of the run
    s ← b
    while b ≠ f_q
          do ▷ invariant: s points to first slot of bucket b
             ▷ skip all elements in the current run
             repeat INCR(s)
                until ¬ is-continuation(A[s])
             ▷ find the next occupied bucket
             repeat INCR(b)
                until is-occupied(A[b])
    ▷ s now points to the first remainder in bucket f_q
    ▷ search for f_r within the run
    repeat  if A[s] = f_r
                then return TRUE
             INCR(s)
       until ¬ is-continuation(A[s])
    return FALSE
```

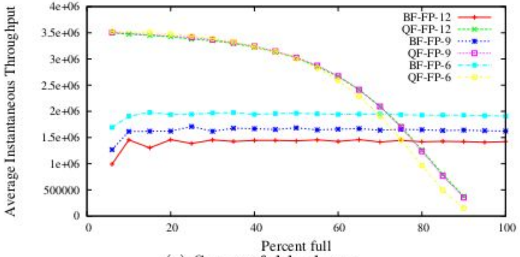
Figure 3: Algorithm for checking whether a fingerprint f is present in the QF A.



(a) Cummulative inserts



(b) Uniform random lookups



(c) Successful lookups

Figure 6: In-RAM Bloom Filter vs. Quotient Filter Performance.

	Capacity	
FP rate	BF	QF (90%)
1/64	1.98 billion	1.71 billion
1/512	1.32 billion	1.29 billion
1/4096	991 million	1.03 billion

Table 2: Capacity of the quotient filter and BF data structures used in our in-RAM evaluation. In all cases, the data structures used 2GB of RAM.

Quotient Filter Pros

- Cache Friendly
- In-order hash traversal
  - Resizing support
  - Merging support
- Delete support
- Robust to FP rate

Quotient Filter Cons

- Smaller Capacity
- Performance degrades when >70% full
- Limited counting support

Rank and Select Quotient Filter (Pandey, et al 2017)

- 2.125+*r* bits/slot vs 3+*r*
- Faster lookups, higher load support
- *rank* and *select* support
  - Optimized with new x86 bit manipulation instructions

RSQF changes

- *occupieds* vector - 1 at position *i* if some *q* in the QF is *i*
- *runends* vector - 1 at *i* if *i* contains last remainder in run
- 1-1 correspondence between both bit vectors

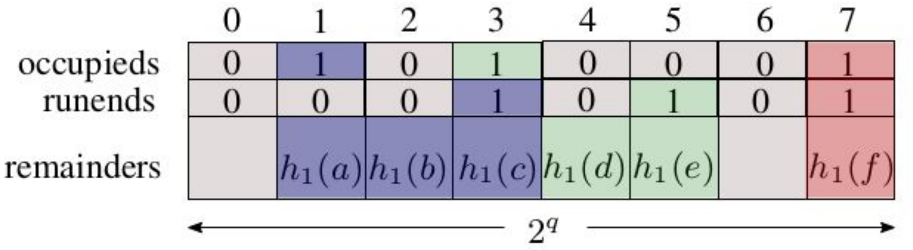


Figure 1: A simple rank-and-select-based quotient filter. The colors are used to group slots that belong to the same run, along with the runends bit that marks the end of that run and the occupieds bit that indicates the home slot for remainders in that run.

Pandey et al 2017

RSQF Query

- hash x to get slot *b*
- find $t = rank_{occupieds} b$
  - (number of runs before *b*)
- Find $l = select_{runends} t$
  - Position of t^{th} runend
- Walk backwards from *l* until either
  - *r* is found
  - Go past *b* or prev runend

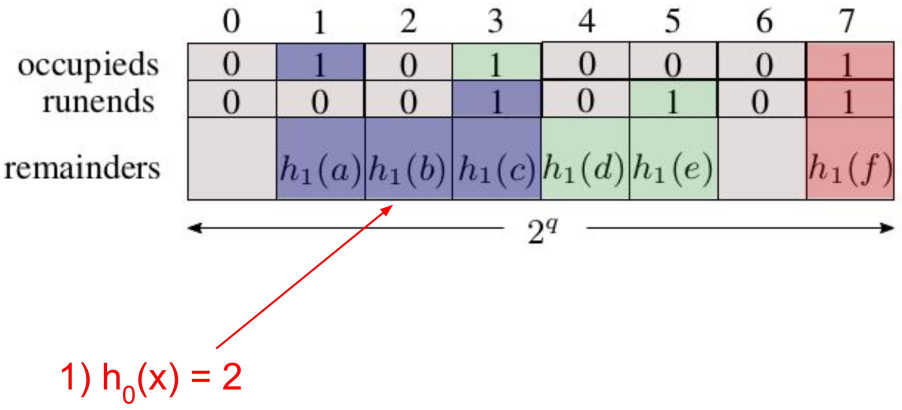
Algorithm 1 Algorithm for determining whether x may have been inserted into a simple rank-and-select-based quotient filter.

function MAY_CONTAIN(Q, x)
    $b \leftarrow h_0(x)$
    if Q.occupieds$[b] = 0$ **then**
        return 0
    $t \leftarrow$ RANK(Q.occupieds, b)
    $\ell \leftarrow$ SELECT(Q.runends, t)
    $v \leftarrow h_1(x)$
    repeat
        if Q.remainders$[\ell] = v$ **then**
            return 1
        $\ell \leftarrow \ell - 1$
    until $\ell < b$ or Q.runends$[\ell] = 1$
    return false

Pandey et al 2017

RSQF Query

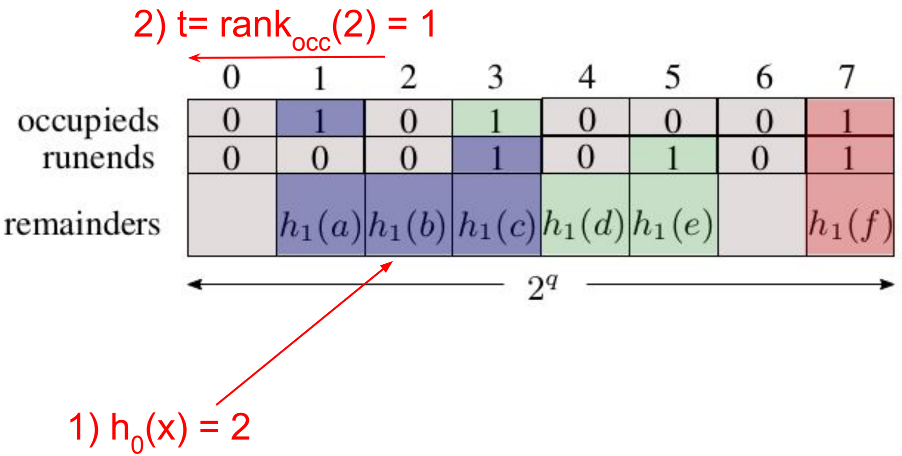
- hash x to get slot *b*
- find $t = rank_{occupieds} b$
  - (number of runs before *b*)
- Find $l = select_{runends} t$
  - Position of t^{th} runend
- Walk backwards from *l* until either
  - *r* is found
  - Go past *b* or prev runend



Pandey et al 2017

RSQF Query

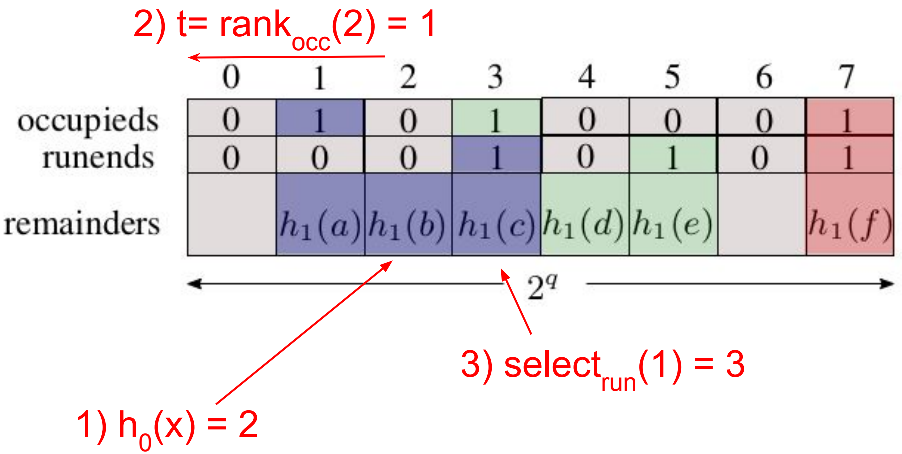
- hash x to get slot *b*
- find $t = rank_{occupieds} b$
  - (number of runs before *b*)
- Find $l = select_{runends} t$
  - Position of t^{th} runend
- Walk backwards from *l* until either
  - *r* is found
  - Go past *b* or prev runend



Pandey et al 2017

RSQF Query

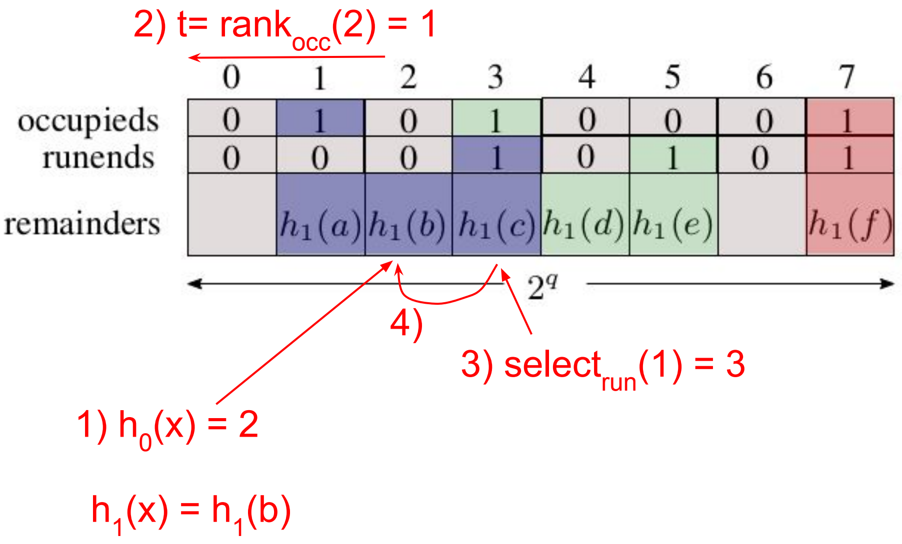
- hash x to get slot *b*
- find $t = rank_{occupieds} b$
  - (number of runs before *b*)
- Find $l = select_{runends} t$
  - Position of t^{th} runend
- Walk backwards from *l* until either
  - *r* is found
  - Go past *b* or prev runend



Pandey et al 2017

RSQF Query

- hash x to get slot *b*
- find $t = rank_{occupieds} b$
  - (number of runs before *b*)
- Find $l = select_{runends} t$
  - Position of t^{th} runend
- Walk backwards from *l* until either
  - *r* is found
  - Go past *b* or prev runend



Pandey et al 2017

RSQF insert

- If *h(x)* is empty, store *r* at *h(x)*
- Else shift everything to the right and insert

Algorithm 2 Algorithm for inserting x into a rank and select quotient filter.

```
function FIND_FIRST_UNUSED_SLOT(Q, x)
    r ← RANK(Q.occupieds, x)
    s ← SELECT(Q.runends, r)
    while x ≤ s do
        x ← s + 1
        r ← RANK(Q.occupieds, x)
        s ← SELECT(Q.runends, s)
    return x

function INSERT(Q, x)
    r ← RANK(Q.occupieds, h0(x))
    s ← SELECT(Q.runends, r)
    if h0(x) > s then
        Q.remainders[h0(x)] ← h1(x)
        Q.runends[h0(x)] ← 1
    else
        s ← s + 1
        n ← FIND_FIRST_UNUSED_SLOT(Q, s)
        while n > s do
            Q.remainders[n] ← Q.remainders[n − 1]
            Q.runends[n] ← Q.runends[n − 1]
            n ← n − 1
        Q.remainders[s] ← h1(x)
        if Q.occupieds[h0(x)] = 1 then
            Q.runends[s − 1] ← 0
        Q.runends[s] ← 1
    Q.occupieds[h0(x)] ← 1
    return
```

Pandey et al 2017

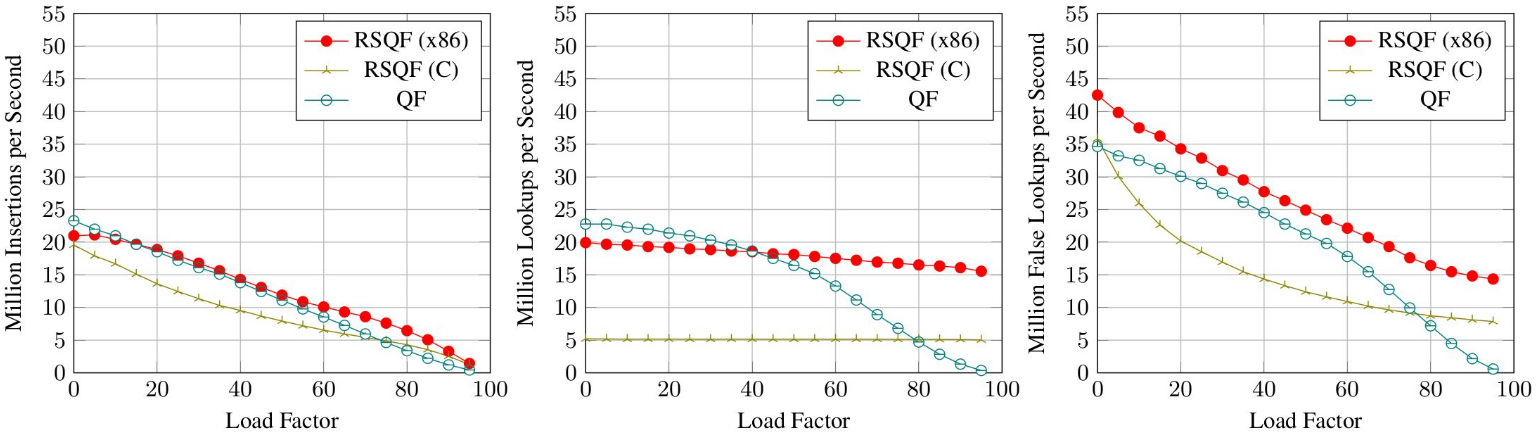
RSQF optimizations

- Offset array *O* - stores distance to runend (or 0 if empy)
- O_i only stored for every 64th slot
  - Can calculate O_j of a slot from nearest O_i using *rank* and *select*
- “Block”-ing array into 64-slot words
  - *Rank* and *select* done on blocks instead of entire array
  - Can use new x86 bit instructions to optimize *rank, select*

offset	occupieds	runends	remainders
8	64	64	$64r$

Figure 3: Layout of a rank-and-select-based-quotient-filter block. The size of each field is specified in bits.

RSQF vs QF



(a) Inserts. (b) Successful lookups. (c) Uniformly random lookups.

Figure 10: In-memory performance of the RSQF implemented with x86 pdep & tzcnt instructions, the RSQF with C implementations of rank and select, and the original QF, all on uniformly random items. The first graph shows the insert performance against changing load factor. The second graph shows the lookup performance for existing items. The third graph shows the lookup performance of uniformly random items. (Higher is better.)

Pandey et al 2017

AMQ -> Counting

- Counting bloom filters
  - Add some bits to the bloom filter
- Countmin sketch
  - Could be more space efficient

Counting Quotient Filter

- Extension of RSQF
- Modifies encoding to support counting

CQF encoding scheme

- Remainders w/ same quotient stored in order
- Counting starts when *remainder*$_i$ < *remainder*$_{i-1}$ in a run
- x ... x = encoding btwn x, x
  - Prepend 0 if encoding > x
- 0 ... 0 0 = encoding btwn 0, 00
- Encoding cannot include x, 0

Count	Encoding	Rules
$C = 1$	x	none
$C = 2$	x, x	none
$C > 2$	$x, c_{\ell-1}, \ldots, c_0, x$	$x > 0$ $c_{\ell-1} < x$ $\forall i \; c_i \neq x$ $\forall i < \ell - 1 \;\; c_i \neq 0$
$C = 3$	$0, 0, 0$	$x = 0$
$C > 3$	$0, c_{\ell-1}, \ldots, c_0, 0, 0$	$x = 0$ $\forall i \; c_i \neq 0$

Table 3: Encodings for C occurrences of remainder x in the CQF.

Pandey et al 2017

CQF Space

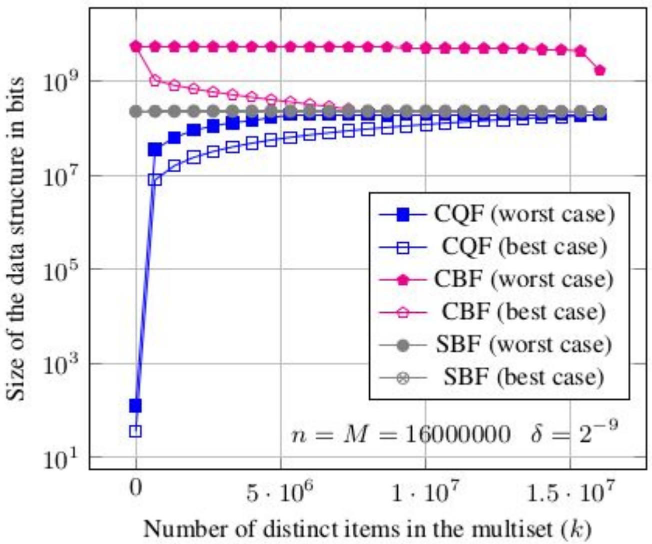


Figure 5: Space comparison of CQF, SBF, and CBF as a function of the number of distinct items. All data structures are built to support up to $n = 1.6\times10^7$ insertions with a false-positive rate of $\delta = 2^{-9}$.

Pandey et al 2017

CQF performance

- Switch to paper

Summary

- Cache efficiency and optimized bit instructions are our friends!
- AMQ with merging, resizing and deletions
- Counting with same amount of space as AMQ

Applications

- Squeakr - kmer counting
  - Replace SBF?
- Other ideas?

Works Cited

Bender, M.A., Farach-Colton, M., Johnson, R., Kraner, R., Kuszmaul, B.C., Medjedovic, D., Montes, P., Shetty, P., Spillane, R.P., and Zadok, E. (2012). Don’t Thrash: How to Cache Your Hash on Flash. ArXiv:1208.0290 [Cs].

Pandey, P., Bender, M.A., Johnson, R., and Patro, R. (2017). A General-Purpose Counting Filter: Making Every Bit Count. In Proceedings of the 2017 ACM International Conference on Management of Data, (New York, NY, USA: ACM), pp. 775–787.

Pandey, P., Bender, M.A., Johnson, R., Patro, R., and Berger, B. (2018). Squeakr: an exact and approximate k-mer counting system. Bioinformatics *34*, 568–575.