Implementation of a Rank and Select Based Quotient Filter

Matthias Bungeroth

February 5, 2018

Abstract

This paper describes a "Rank und Select Based Quotient Filter"(RSQF) and it's implementation. The RSQF consists of two meta-data arrays and a remainder array and has linear runtime. Moreover, the runtime of insert and query can be improved by using offsets and the cache efficiency can be increased by splitting the data into blocks. The RSQF is easy to expand to a Counting-RSQF(CQF) which allows efficient counting of elements inserted. The implementations of the RSQF and the CQF are compared to a basic "Bloomfilter"(BF) and "Counting Bloomfilter"(CBF). The RSQF and CQF have better runtime compared to the BF and the CBF due to better cache performance and fewer lookups in the table.

1 Introduction

This paper will show the advantages of a "Rank und Select Based Quotient Filter"(RSQF) compared to a Bloomfilter(BF). It will show that the RSQF can be implemented in a cache efficient way, that makes the RSQF faster than the BF. In addition, it will show that the RSQF can be expanded to a Counting-RSQF(CQF) which is more space efficient than a Counting-BF.
The paper explains the properties and the usage of a Filter, first.
The second section describes how a basic RSQF works and how it is represented in memory.Furthermore, it will turn linear runtime into constant runtime and increase the cache efficiency. The end of the section describes how efficient counting of elements can be added to the RSQF.
The next section explains the implementation of the RSQF and the CQF.
The last section evaluates the RSQF and CQF by comparing them with a BF and a CBF.

2 Filters in general

This section will explain what filters are and what they are used for.

2.1 Filter

Filters are data structures comparable to a set. They implement operations such as "insert" and "query". Both operations should have runtime of $O(\log n)$ or $O(1)$.The operation "insert" adds elements to the filter, whereas "query" returns true if an element is contained in th filter. If an element x was not inserted into the filter, "query" will return false with a probability of $(1-\delta)$, where δ is the false-positive-rate of a filter, which can be defined during creation. The smaller the false-positive-rate δ, the lager the space consumption of the filter.

2.2 Counting filter

Counting-filters are data structures similar to filters. The only difference being "query" returning the number of elements inserted in the Counting-filter, instead of true or false. The result is greater than the real count with a portability of $(1-\delta)$, but never less.

2.3 Usage of filters

Filters are usually used if fast access time and low space consumption are required, whereas the error δ is negligible for the use-case. For example, a pre-selection of elements can be done efficiently. If query returns false, the database does not have to be searched.

3 Rank and Select Based Quotient filter

The following will explain how a Rank and Select Based Quotient filter works.

3.1 Architecture

The following will describe how elements are saved and represented in the filter, as well as they are inserted and queried.

3.1.1 Memory-representation

A Rank and Select Based Quotient filter owns a hash-function $h : X \to \{0, \dots, 2^p - 1\}$, that projects onto a natural number, that consists of $p = \log_2 \frac{n}{\delta}$ bits. $h(x)$ is divided into $h0(x)$ the quotient and $h1(x)$ the remainder. $h0(x)$ consists of the upper $(p-r)$ bits and $h1(x)$ of the lower r bits of $h(x)$. (r can be configured in relation to p.)
To save remainders (h1), the RSQF maintains an array of length 2^q with r-bit slots. By inserting an element x the remainder $h1(x)$ will be saved in its homeslot $h0(x)$. If the slot is already in use by another remainder, the collision will be resolved by using a variant of linear probing. Next, this variant will be explained.
In order to guarantee that an element is saved in its home slot (if this is possible) and remainders of elements with the same quotient are stored in consecutive slots, the filter needs some meta-data and invariants.

Meta-Data:

- occupied array of length 2^p with 1-Bit Slots
- runend array of length 2^p with 1-Bit Slots

Invariants:

- occupied$[x] = 1 \iff \exists_{y \in S} : h0(y) = x \iff$ slot x is occupied
- runends[b]=1 $\iff$ slot b contains the last remainder in a run.
- $\forall_{x,y \in S} : h0(x) < h0(y) \implies h1(x)$ is stored in an earlier slot than $h1(y)$
- If $h1(x)$ is stored in slot s, then $h0(x) \leq s$ and there are no unused slots between slot $h0(x)$ and slot s, inclusive.

S is a set of elements that have been inserted into the filter and a slot is unused if no remainder is stored in it.
These invariants imply that for each set bit in occupied there exits exactly one set bit in runend. An example can bee seen in figure 1.

slot	0	1	2	3	4	5	6	7
occupied	1	1	0	0	1	0	0	1
runend	0	0	0	1	1	1	0	1
remainders	$h1(a)$	$h1(b)$	$h1(c)$	$h1(d)$	$h1(e)$	$h1(f)$	0	$h1(g)$

Figure 1: Elements a-g have been inserted into this RSQF. $h0(a) = h0(c) = h0(b) = h0(d) = 0$, $h0(e) = 1$, $h0(f) = 1$, $h0(g) = 7$

3.1.2 Operations

When inserting or querying an element x, the end of the corresponding run starting at homeslot $h1(x)$ has to be found. This is equal to finding the runend-bit associated with the bit occupied[$h1(x)$]. This can be done with the operations $RANK$ and $SELECT$.

$$RANK(B, i) = \sum_{j=0}^{i} B[j] \text{ (Ammount of set bits in B up to postion } i\text{)}$$

$$SELECT(B, i) = \text{(Index of the } i\text{th set bit in B)}$$

$h1(x)$, $d = RANK(occupied, h1(x))$ counts the number of set bits in occupied up to position $h1(x))$. $s = SELECT(runend, d)$ finds runend bit s associated with the homeslot $h1(x)$.
This can be simplified to the operation rankSelect(x) which finds the corresponding runend to a slot if it exists.

$$rankSelect(h0(x)) = SELECT(runend, RANK(occupied, x))$$

For all y > z = (Index of the earlier set bit in occupied of slot y) it applies that: $occupied[y] = 0 \implies rankSelect(y) = rankSelect(z)$

Insert To insert an element x in the RSQF the following should be done:
s = rankSelect(h0(x)) finds the associated runend bit s.
If $s < h0(x)$, slot $h0(x)$ is unused and can be used. The following should be set: occupied[h0(x)]=1, runend[h0(x)]=1 and remainders[h0(x)] = h1(x).
if $s \geq h0(x)$, the next unused slot n must be found. After that, all remainders and runend bits between $(s + 1)$ and $(n - 1)$ must be shifted one by one to the slots between $(s + 2)$ and (n). Finally the following should be set: occupied[h0(x)]=1, runend[s+1]=1 and remainders[s+1] = h1(x). If occupied of[$h0(x)$] was already set before the insert operation, runend[s] should be set to 0.

Query Figure 2 shows the query algorithm returning true if x was inserted in the RSQF.

3.2 Improvement of runtime by offsets

Insert and query have a runtime of O(2^q), because rankSelect has a runtime of O(2^q). To improve the runtime to O(1) offsets should be added. An offset saves the distance between an occupied bit and its associated runend bit. Furthermore, O_i is defined by $rankSelect(i) - i$ and is only the defined if and only if occupied[i]=1, otherwise the offset O_i is not well defined. An offset O_j can be computed by a defined offset O_i if $i < j$.
To save space, the offset will only be saved for every 64th slot. To ensure that all

```
s = rankSelect(h0(x))
do{
   if remainders[s] = h1(x) then
      return true;
   s = s-1;
}while(s>h0(x) and !runend[s]);
return false;
```

Figure 2: This algorithm checks if the associated remainder in slot $\leq s$ was found, until s < Homeslot h0(x) or the next runend bit of another run was found.

```
rankSelect( h0) :
   i = ((h0 / 64) * 64);//Index of next lower offset
   if i = h0:
      return offset[i / 64] + h0
   else:
      d = RANK(occupieds, i + 1, h0 - i - 1)
      if d = 0:
         return i + offset[i / 64]
      else:
         t = SELECT(runends, i + offset[i / 64] +1, d)
         return i + offset[i / 64] + t +1
```

Figure 3: This algorithm shoqs the rankselect operation with offsets.

offsets are well defined, each 64th runend and occupied bit will be set to 1. To check if an element was already inserted into a 64th slot, the RSQF has to maintain another array "used" of length $\frac{2^p}{64}$ with 1-bit slots.

$used[i/64(integerdivision)] = 1 \iff$ an element x with h0(x)=i was inserted into the filter

As a consequence of using offsets, rankSelect can be redefined with a time complexity O(1) as figure 3 shows. Rank and select are defined by:
$RANK(B, a, b) = RANK([B_a, B_{a+1}, \ldots, B_{2^p}], b)$

$SELECT(B, a, b) = SELECT([B_a, B_{a+1}, \ldots, B_{2^p}], b)$

If RANK returns 0, it is obvious that select will return zero and no additional step between (i + offset[i / 64]) and (i + offset[i / 64] +1) has to be added.

Because (h0 - i - 1) is usually small compared to 2^p and an associated runend bit is usually not "far" away this algorithm has time complexity O(1).
These offsets can be updated whenever an insert operation shifts runend bits.

3.3 Improvement of cache-efficiency by blocking data

Insert and query are not cache efficient because there are a couple of different lookups in the occupied, runend and offset array. This is not cache friendly because arrays have a minimum distance of $\frac{2^p}{bytes}$ bytes. To make the RSQF cache efficient the occupied, runend and offset array will be broken down into blocks. Each block owns 64 entries. The block's structure is described in figure 4.
This structure is cache efficient because if an element is inserted, only consecutive blocks have to be observed (ordered).

7	1	64	64	$r \cdot 64$
offset	used	occupieds	runends	remainders

Figure 4: Representation of one block

Count	Encoding	Rules
1	x	none
2	x, x	none
	For $x \neq 0$	
>2	$x, c_{l-1}, \ldots, c_0, x$	$x > 0$
		$c_{l-1} < x$
		$\forall_{i<l-1}\ c_i \neq x$
		$\forall_{c_i} \neq x$
	For $x = 0$	
3	x, x, x	none
>3	$x, c_{l-1}, \ldots, c_0, x, x$	$\forall_{c_i} \neq x$
		$\forall_{i<l-1}\ c_i \neq x$

Figure 5: Shows encoding of counters for an element x

3.4 Counting

The RSQF counts elements unary. By inserting an encoded counter instead of storing the same remainder a couple of times, the amount of used slots in the filter will be reduced. Counters can be found by storing elements in increasing order. To store a counter for a remainder x this counter has to violate the increasing order. That implies the first symbol of a counter has to be smaller than x. To mark the end of a counter the remainder x will be stored again beyond the counter. That implies that the counter is not allowed to have a symbol that equals to x.
If $x = 0$, it is not possible to start a counter by using a smaller symbol. A counter for $x = 0$ can be identified by two trailing zeros. That means for each run only two consecutive zeros are allowed, because they mark the end of a counter for the remainder "0". This implies zeros can not be used as symbols for encoding counters. Counters are encoded as shown in figure 5.

For $x \neq 0$ and count $C \geq 3$:
C can be encoded as $C - 3$ as $c_{l-1}, \ldots, c_0$ in base $2^r - 2$ where symbols are $1, 2, \ldots, x-1, x+1, \ldots, 2^r - 1$ and add a trailing zero if $c_l >= x$.

For $x = 0$ and count $C \geq 4$:
C can be encoded as $C - 4$ as $c_{l-1}, \ldots, c_0$ in base $2^r - 1$ where symbols are $1, 2, \ldots, 2^r - 1$.

Using counters in a RSQF demands that $r \geq 2$ because otherwise the base would be zero.

4 Implementation

Hereafter, the implementation of the Rank and Select Based Quotient filter and the Counting Rank and Select Based Quotient filter will be explained.

4.1 Rank and Select Based Quotient Filter

First, the implementation of the Rank and Select Based Quotient filter will be explained.

4.1.1 Blocks

The filter saves the data described as in section 3.3. The block struct is defined in "Block.h":

```
struct Block{
  uint8_t offset:7;
  uint8_t used:1;
  uint64_t occupied;
  uint64_t runend;
  uint64_t remainder[REMAINDER_LEN];
}__attribute__((packed));
```

The first 8 bits are divided into 7 offset bits and 1 used flag (introduced in 3.2). 7 offset bits allow a maximum offset of 127, which is usually enough.
(REMAINDER_LEN = r, can be changed in Properties.h. r can be $2^n, n \in \mathbb{N}$).
The attribute "packed" guarantees that the compiler does not add padding to save memory.

4.1.2 Rank und Select

Rank is only called on occupied and select only on runend. The rank and select operation expects a reference to a block and a starting bit in that block to call rank or select on.

Rank For an efficient implementation of the operation rank, "*__builtin_popcountll*" can be used. It returns the number of set bits in a 64Bit integer. To use this method 64Bit integers have to be extracted out of consecutive blocks. This can be done as following:

```
  uint64_t r = ((blocks[i].occupied << startingBit)
                  |(blocks[i+1].occupied >> (64 - startingBit))
```

Then rank(r , min(number of bits still to consider, 64)) gets called until the correct count of bits to consider is zero. The sum of all rank operations will be returned. rank(r,n) sets the least significant (64-n) bits to zero and then calls popcountll.

Select Select(B,i) inspects each bit in a loop and returns the index of the ith set bit.

RankSlect The rankSelect operation was implemented as described in section 3.2. Furthermore, it calculates the bit index inside a block to call rank and select.

4.1.3 Query

The rankSelect operation was implemented as described in section 3.1.2. It is not possible to access a remainder directly, it can be extracted out of a block with the method ”getRemainder”.

4.1.4 Insert

The insert operation was implemented as described in section 3.1.2, with slight changes due to the adding of offsets and blocking. Each offset pointing to a set runend bit between the result of rankSelect and the next free slot will be increased by one.

4.2 Counting Rank und Select Based Quotient Filter

The CQF implementation encodes the counter as described in section 3.4. The implementation reverses the encoding of the counter.

4.2.1 Changes in Query

The operation query tries to find the correct remainder while skipping counters of different remainders. If the correct remainder was found, it decodes the corresponding counter and returns the count.

4.2.2 Changes in Insert

Insert tries to find a counter, for an element to insert, in the same way that query does.
If the element does not exist, the element will be stored at the correct position, so that all remainders in one run are sorted in descending order.
If the element exists, the counter will be incremented. This may need one more unused slot which can be gathered by shifting runends and remainders.

5 Evaluation

All testes have been executed on a machine with Intel-Core i7 4820k processor and 16GB-DDR3-RAM.
A filter is always configured with the given false-positive-rate δ and the number of elements n to insert.

5.1 Rank and Select Based Quotient Filter

The test data is created randomly (by c++ ”rand” function)). An element consists of a random long with the identity function as hash function. It is very likely that test data of n elements is distinct.

5.1.1 Comparison of RSQF variants

Table 1 shows speed tests for the three variants of the RSQF.
The RSQF with no blocking (nb) is the slowest because of linear runtime of the rankSelect operation. The speedup by factor of 3 from nb to the normal RSQF is caused by the cache efficient implementation and the overhead of ”std::vector” which was used in the nb implementation .

Configuration	Operations	RSQF no	RSQF nb	RSQF
$\delta = 0.001$	Random insert	20s	<5ms	<2ms
$n = 10000$	Query on inserted elements	20s	<5ms	<2ms
	Random query(100% load)	0.1s	<1ms	<0.5ms
$\delta = 0.0001$	Random insert	/	1.4s	3.7s
$n = 10000000$	Query on inserted elements	/	1.8s	5.3s
	Random query(100% load)	/	0.7s	0.6s
$\delta = 0.001$	Random insert	/	43s	15s
$n = 100000000$	Query on inserted elements	/	52s	17s
	Random query(100% load)	/	8.2s	7.3s

Table 1: Speedtest for RSQF with no offsets (no), the RSQF with no blocking (nb) and the RSQF with offsets and blocking

Configuration	**Operations**(in million per second)	BF	RSQF
			(r=4)
$\delta = 0.01$	Random insert	2.9	6.0
$n = 10000000$	Query on inserted elements	3.2	7.6
	Random query(100% load)	12.7	12.0
			(r=8)
$\delta = 0.00001$	Random insert	1.6	8.8
$n = 10000000$	Query on inserted elements	1.8	6.6
	Random query(100% load)	12.26	25.7
			(r=16)
$\delta = 0.000001$	Random insert	1.1	4.7
$n = 100000000$	Query on inserted elements	1.3	5.0
	Random query(100% load)	10.0	10.4

Table 2: This table compares the BF and the RSQF for different δ and n

5.1.2 Comparison with Bloomfilter

Runtime The runtime grows for larger remainders because longer remainders have to be compared or shifted. The RSQF's increases with lower δ (because of less collisions) whereas the BF gets slower (because more hash functions are used). Random queries are fast in BF because only a few hash functions are considered. This can be observed in table 2.

In addition, the BF is capable of performing inserts and lookups of inserted elements in constant time (independent from the load factor). Random lookups in the BF get worse with increasing load factors because more slots of different hash-functions must be checked.

Random inserts in the RSQF need more time with increasing load-factor, due to the time consuming resolving of collisions. Random lookups require more time because a whole run has to be checked before returning false if possible. This can be observed in figure 6.

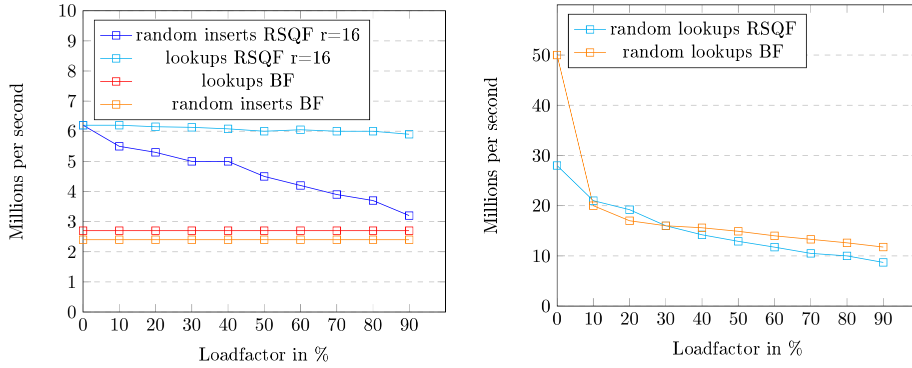

Figure 6: Graphs showing runtime for different load factors ($n = 100000000$, $\delta = 0.001$)

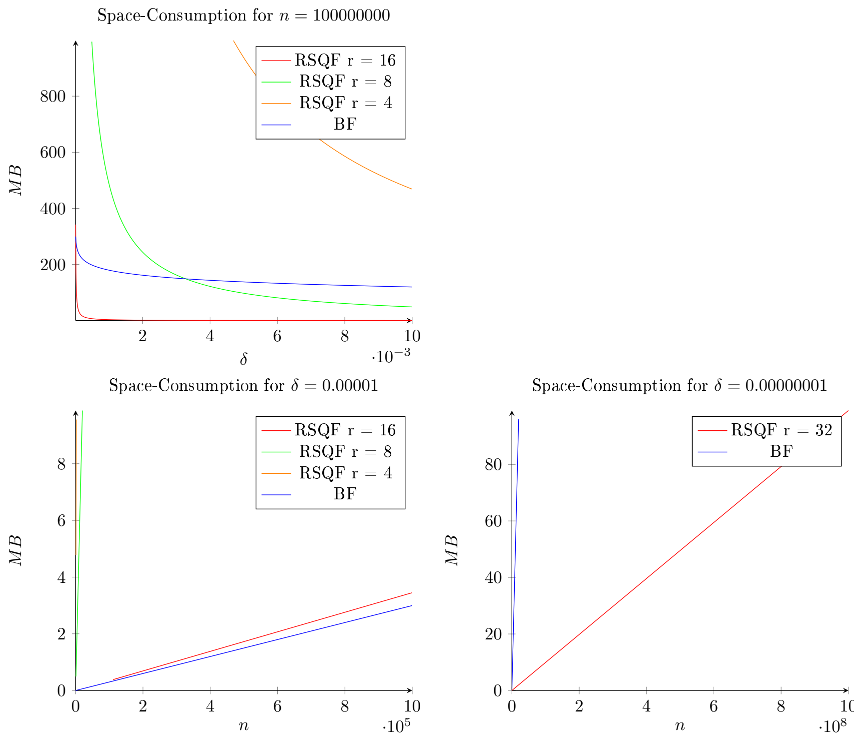

Figure 7: Graphs showing the relationship between r , n and δ

Operations(in million per second)	**CBF**	**CQF**(r=8)
Random insert	7.9	13.5
Random lookup	7.7	9.6

Table 3: Table showing average runtime of 1 000 000 000 operations each for a CQF/ CBF configured with $\delta = 0.0001$,$n = 2000$

Space-analysis The needed space can be calculated (in Bit):

$$bf = \frac{-n \ln \delta}{\ln 2 \ln 2} \quad rsqf = 2^{\log_2(\frac{n}{\delta})-r}(2 + r + \frac{8}{64})$$

Where $2^{\log_2(\frac{n}{\delta}-r)}$ is the number of slots. In the equation the 2 stands for the occupied and runend bit vector, the r for remainders and $\frac{8}{64}$ for the offsets. Figure 7 shows the relations between r , n and δ.

The larger r the lower the space consumption. This implies r is the factor to choose between runtime and memory usage. The lower r, the better the runtime and the more memory is used. The larger r, the better the memory usage and the larger the runtime.

5.2 Counting Rank und Select Based Quotient Filter

The test data is created as in section 5.1. n will be amount of distinct elements inserted.

Runtime For distinct insertion sets the runtime of the CQF and the RSQF does not vary for any operation because no counters are needed. For distinct sets the CBF performs worse than the BF .
Performance of CQF and CBF can bee seen in table 3.
If n grows, the performance of the CQF will stay the same, while the one of the CBF will get worse, because more slots have to considered every insert or lookup.

Space-analysis While the CQF does not need more space than the RSQF, the CBF needs c (counter size in bits) times more space than the BF. The space consumption of the CBF gets even worse while the memory usage of the CQF stays the same (compared to the non counting versions). c should be 16, 32 or 64 depending on the amount of elements to insert.

6 Conclusion

It was shown that a RSQF can be implemented cache efficient with runtime O(1). The RSQF was easy to expand to a CQF that never uses more slots than the RSQF due to efficient counter encoding. The RSQF/CQF can have better runtime and memory usage than a BF/CBF. The relationship of runtime and memory usage of the RSQF/CQF can be configured by setting the remainder length to the correct value.

Rank and Select Based Quotient filters are good alternatives to Bloomfilters because of better runtime up to a factor of 4. Only random lookups in empty filters are slow, but this is a scenario that usually does not happen.
This implies running Rank and Select Based Quotient filters is always better than running Blommfilters.

References

[1] Prashant Pandey, Michael A. Bender, Rob Johnson, and Rob Patro. A General-Purpose Counting Filter: Making Every Bit Count