Bloom Filter and Variation

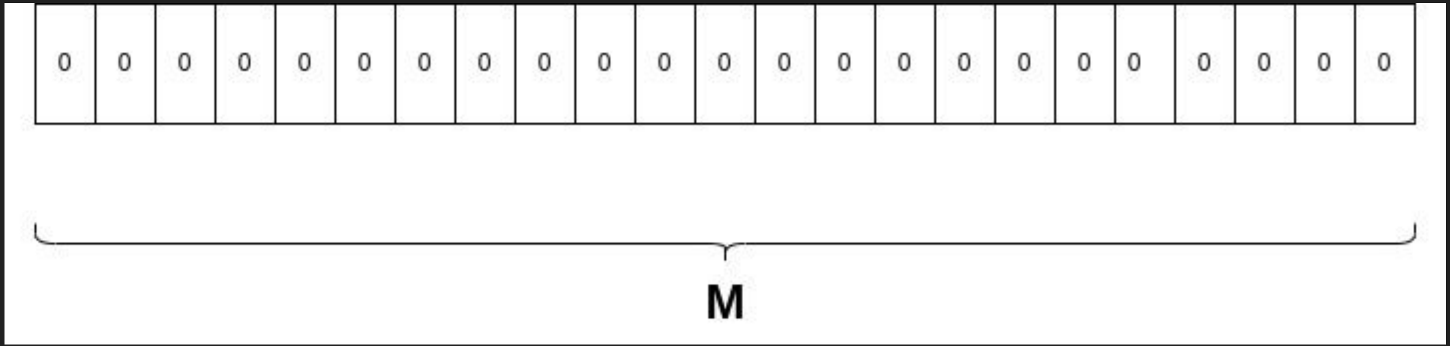
Shicheng Fang

Bloom Filter: What is it?

- Test an element exists in set
- Support two operations:
  - **add(key)**
  - **isExist(key)**
- O(1) Time and Space complexity
- Possible erroneous result for isExist(key)
  - isExist(key) returns true though key was not in the set
  - We call it **False Positive**

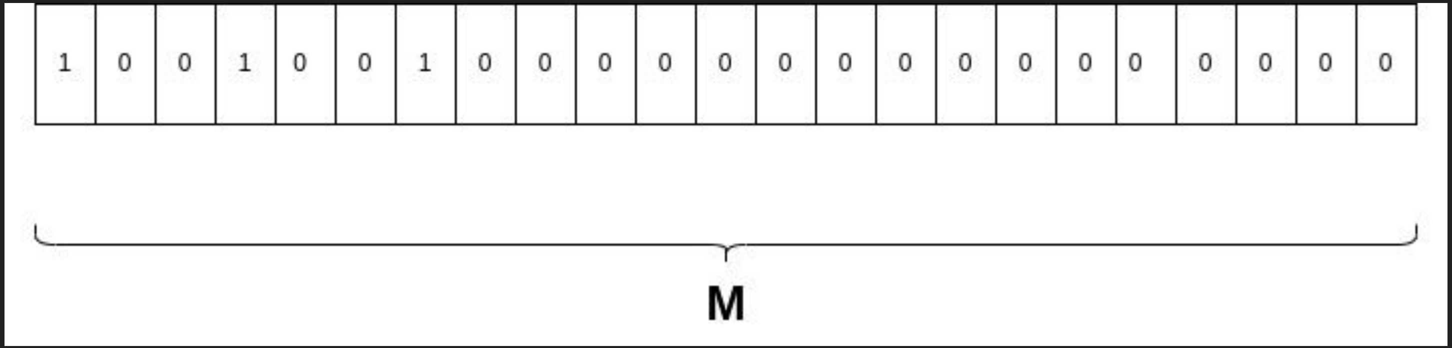
Bloom Filter: How does it work?

- Initially, an empty bit array filled with 0 of size M
- M is usually a large integer



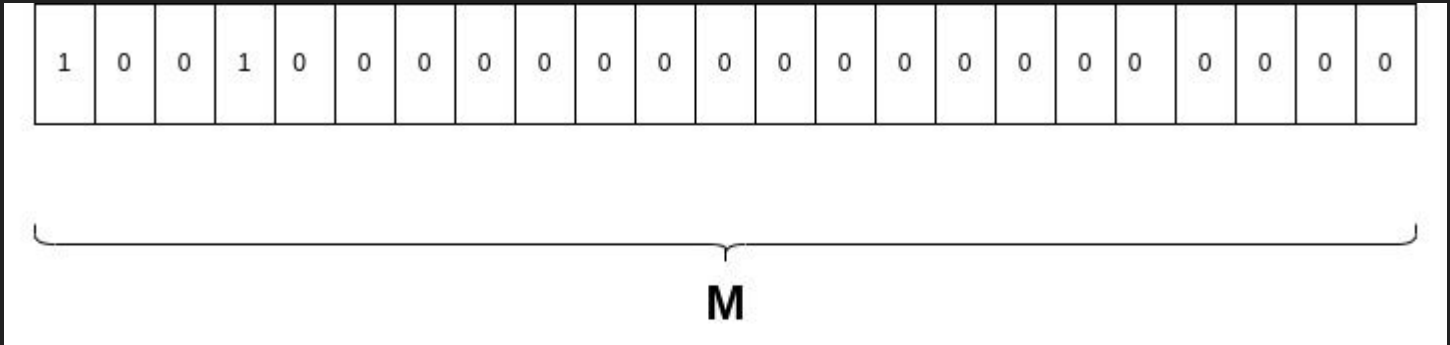
Bloom Filter: How does it work? (continue)

- Add(key)
  - K hash functions calculate K values as array index
  - The K array indexes will be filled with 1
- Example:
  - K=3
  - hash1(key)=0, hash2(key)=3, hash3=6



Bloom Filter: How does it work? (continue)

- isExist(key)
  - K hash functions calculate K values as array index
  - Return if all K locations are 1
- Example:
  - K=3
  - hash1(key)=0, hash2(key)=3, hash3=6
  - Array[6] != 1, return false



Bloom Filter: False Positive

- isExist(key) returns true even if key was not added before
- Consider Bloom Filter filled with 1
  - isExist(key) ALWAYS returns true, not useful at all

Bloom Filter: Tradeoff

- Tradeoff between:
  - False Positive rate (fp)
  - Bloom filter size (M)
  - Number of items in the filter (N)
  - Number of hash functions (K)
- In general,
  - K=ln(2)*M/N round to integer
  - Higher M => Lower fp
  - Higher N => High fp

Bloom Filter: Performance Tuning

- Calculate index using bitwise AND instead of %
  - Bitwise AND is faster than %
  - A%B == A&(LOG2(B)-1)
  - B is power of 2
- Calculate hash for once instead of k
  - Base, Inc = hash_function(key)
  - Set Base + 1 * Inc, Base + 2 * inc, …, Base + k * inc

What if M is not power of 2?

- Enforce M to be power of 2 (Inflexible)
- Use % (Low performance)
- Turn % into a sequence of faster operations
  - Usually, sequence is Multiply + Shift + Add/Sub

Multiply, Shift, Add

- Common technique in compiler optimization of %
- However, M is unknown at compilation time
  - Compiler cannot optimize %
- Solution:
  - Make M available at compilation time
    - Need recompilation when M changes
    - Very inflexible
  - Perform optimized % at runtime

Multiply, Shift, Add

- A%B=A-(A/B)*B
- A/B takes most clock cycles
    - How to optimize division?

Faster division

E.g. Divide by 3

```
computeDivideBy3(long x) {
        long y = x;
        y = y * 1431655766;
        y = y >> 32;
        x = x >> 31;
        return y - x;
```

- Technique called **division by reciprocal approximation**
- 1431655766 is magic number.
- **EVERY** divisor has its own magic number.

What is 1431655766?

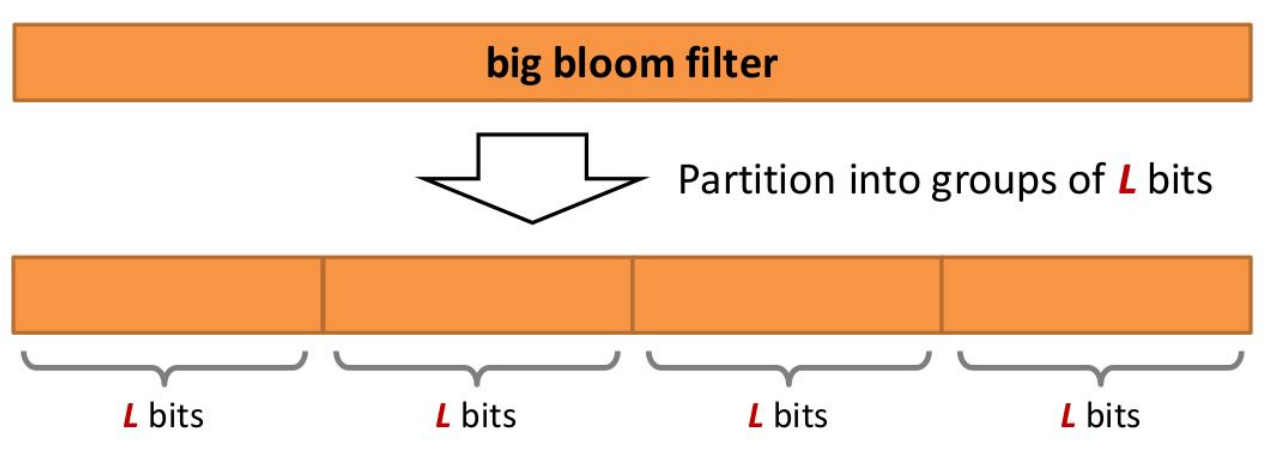
- Convert the divisor 3 to its reciprocal, which is 1/3.
- Multiply the reciprocal by a large power of 2, e.g. 2^30
  - 1/3 * 2^30 = 1073741824 / 3
- Round the resulting value to the nearest integer, which gives us 1431655766.
- This value is precalculated for once and stored for later use.

MOD or Multiply, shift, add

- Which is faster? (On Pentium CPU)
    - DIV takes 40 clock cycles.
    - MUL takes 10 clock cycles.
    - Add/Sub takes 1 clock cycle
    - Multiply, shift, add is significantly faster than MOD
- On other platform, MOD may be faster

Bloom Filter: Blocked Filter Variation

- Observation:
  - Set K bits can cause at most K cache miss
- Block Filter:
  - Divide the array into multiple cache lines each with size L bits, L=512
  - Set k bits in one cache line
  - Only 1 cache miss



Source:
Design Tradeoffs of
Bloom Filters
Jianguo Wang (UCSD)

Blocked Bloom Filter Disadvantage

- Higher FP rate (Bits are squeezed into a cache line)
- K bits should be set separately (K writes are needed)

Bloom Filter: Register Bloom Filter Variation

- Observation:
  - Set K bits need K write (K MOV instructions)
- Register Filter:
  - Divide the array into blocks with size of a register (64 bits) rather than blocks (512 bits)
  - K bits are set simultaneously by constructing the mask and then doing OR :
    - E.g. to set index 0, 3, 5, and 6, only OR 101011
  - MOV is executed for once

Register Bloom Filter Disadvantage

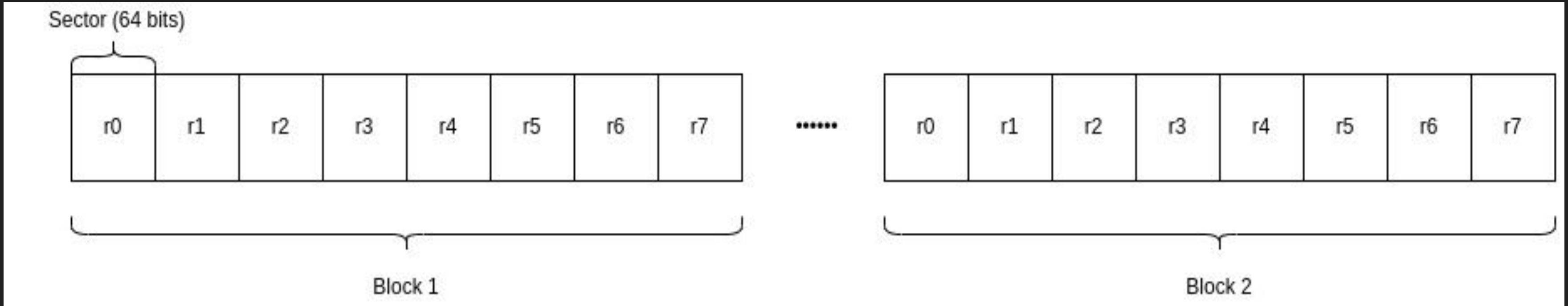
- Even higher FP (K bits are squeezed in 64 bits)

Bloom Filter: Sectorized Filter Variation

- Blocked Bloom Filter: too slow
- Register Bloom Filter: high fp
- Can we make a tradeoff?
    - Sectorized Filter

Bloom Filter: Sectorized Filter Variation

- An extension of Blocked Bloom Filter
- One block is divided into multiple into sectors (8 sectors as shown)
- To set K bits, K bits are set in C sectors

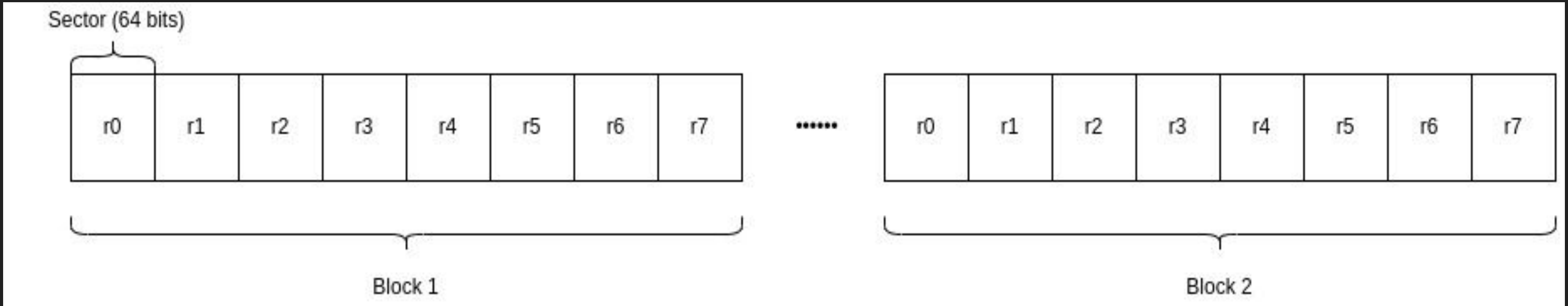


Bloom Filter: Sectorized Filter Variation (cont.)

Example:

8 sector, each with size 64 bits

C=2, K=8

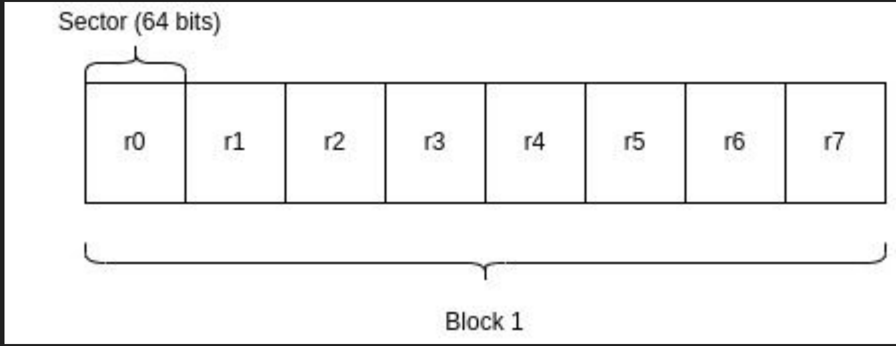


Bloom Filter: Sectorized Filter Variation (cont.)

Example: (cont.)

K=8, C=2, meaning 2 sectors are used, e.g. r0 and r1.

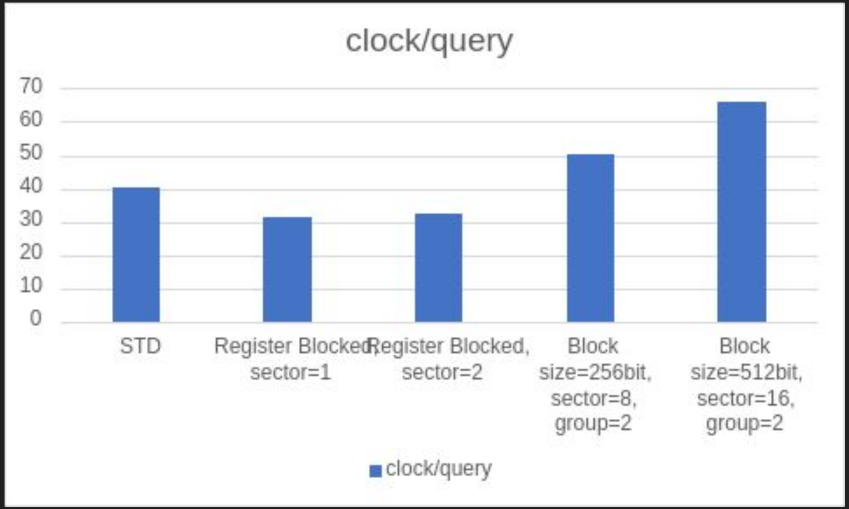
4 bits set in each sector, 4 bits can be set simultaneously like a register block bloom filter

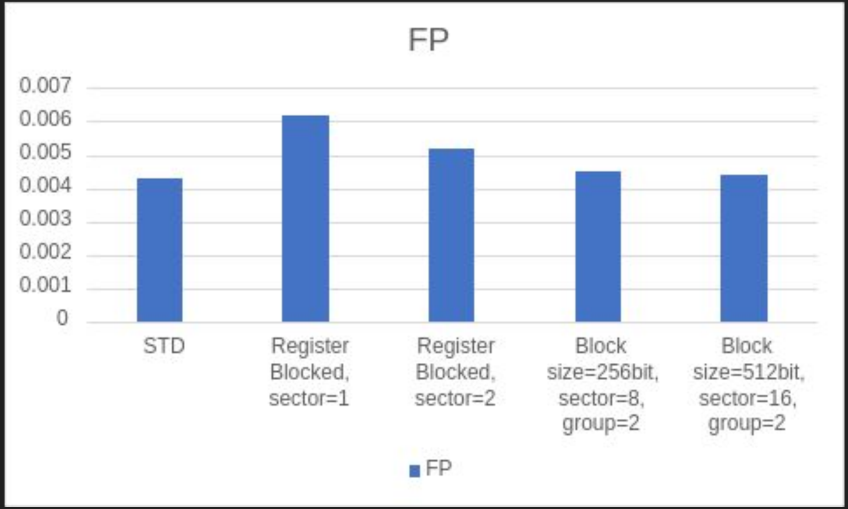


Performance Comparison

STD is the standard bloom filter used in Cassandra and RockDB

n = 147639508
m=1181116006 bits (too large, stored in L3/dram)
k=2





Small size favors register blocking

n = 132416
m=1059061 bits (L1)
K:8