Bloom Filter Compression for In-Memory Databases

Jianguo Wang (UCSD)

Mentors:
Dr. Garret Swart
Dr. Weiwei Gong
Bloom Filter Background

<table>
<thead>
<tr>
<th>Key</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>k1</td>
<td>v1</td>
</tr>
<tr>
<td>k2</td>
<td>v2</td>
</tr>
<tr>
<td>k3</td>
<td>v3</td>
</tr>
<tr>
<td>k4</td>
<td>v4</td>
</tr>
</tbody>
</table>

1% YES (false positive)

100% YES

99% NO
Bloom Filter Background

Element $e$

- $H_1(e)$
- $H_2(e)$
- $\cdots$
- $H_k(e)$

**bit vector**

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td></td>
<td></td>
<td>1</td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

$n$: number of elements

$m$: bloom filter size (in bits)

$k$: number of hash functions

$p$: false positive ratio (FPR)

$$p = \left(1 - e^{-kn/m}\right)^k$$
How to design a bloom filter to achieve the best tradeoffs?
Design Tradeoffs

Element $e$

$H_1(e)$  $H_2(e)$  ...  $H_k(e)$

1 1 1...

time

e.g., $k = 1$

space

bigger $k$ compression

false positive

more space

bigger $k$
Algorithms

Implemented different algorithms to understand tradeoffs

<table>
<thead>
<tr>
<th>algorithms</th>
<th>$k$ (# hash functions)</th>
<th>compressed or not</th>
<th>cache-aware or not</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base1</td>
<td>1</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>BaseK</td>
<td>2, 4, 8</td>
<td>N</td>
<td>N</td>
</tr>
<tr>
<td>CacheK</td>
<td>2, 4, 8</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>MComp</td>
<td>1</td>
<td>Y (mask-based)</td>
<td>Y</td>
</tr>
<tr>
<td>PComp</td>
<td>1</td>
<td>Y (position-based)</td>
<td>Y</td>
</tr>
</tbody>
</table>
Algorithm: Base1

Element $e$

$H(e)$

pos = $h \% \text{bf\_size}$;
result = $bf[\text{pos} / 8] \& (1 << (\text{pos} \& 8))$;

pos = $h \& ((1 << \log_2 \text{bf\_size}) - 1)$;
result = $bf[\text{pos} >> 3] \& (1 << (\text{pos} \& 7))$;
Algorithm: BaseK

Use $k = 4$ hash functions

Element $e$

Problem: How to generate the $k$ positions?

<table>
<thead>
<tr>
<th>Solution 1: call $k$ different hash functions</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>less collision $\Rightarrow$ better false positive</td>
<td>slow</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Solution 2: call 1 hash function and produce $k$ positions</th>
<th>Pros</th>
<th>Cons</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>fast</td>
<td>slightly worse false positive</td>
</tr>
</tbody>
</table>
h = query_keys[idx];
delta = (h >> 34) | (h << 30);  // Rotate right 34 bits
mask = (1 << log2_bf_size) - 1;

// generate k positions based on one hash value
bitpos0 = h & mask;  // h % bf_size
bitpos1 = (h + delta) & mask;
bitpos2 = (h + delta + delta) & mask;
bitpos3 = (h + delta + delta + delta) & mask;

result =
((bf[bitpos0 >> 3]) & (1 << (bitpos0 & 7)))
&
((bf[bitpos1 >> 3]) & (1 << (bitpos1 & 7)))
&
((bf[bitpos2 >> 3]) & (1 << (bitpos2 & 7)))
&
((bf[bitpos3 >> 3]) & (1 << (bitpos3 & 7)));

// generate k positions

whether all bits are 1

<table>
<thead>
<tr>
<th>time (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>logical and (&amp;&amp;)</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>k = 2</td>
</tr>
<tr>
<td>20</td>
</tr>
</tbody>
</table>
Algorithm: CacheK
(Cache-aware bloom filter)

Main idea:
- Partition a big bloom filter to many cache lines
- Restrict the $k$ positions of every element to a cache line
- Goal: at most 1 cache miss per element

Element $e$
probe
decide which cache line

512 bits
(cache line)

[FPJ09] Cache-, hash-, and space-efficient bloom filters
[LWL+17] Ultra-Fast Bloom Filters using SIMD Techniques
Algorithm: MComp
(Mask-based compression)

If the bloom filter is **sparse** enough, can we **compress** the bloom filter?

If the bloom filter is sparse enough, can we compress the bloom filter?

```
00010000010000000000000000000000000000000000000000000000000000000000000000000000000...
```

64 bits: two 1's

```
0 0 0 0 1 1
```

&

```
0 0 1 0 0 1
```

AND_MASK = 0000001

```
0 0 0 0 1 1
```

OR_MASK = 0010011

Compressed size: 6 + 6 = 12 bits (originally 64 bits)

**Query:** is the $i$-th position 1?

\[
\begin{align*}
i \& \text{ AND MASK} & = & \text{AND MASK} \\
\text{OR MASK} & = & \text{OR MASK}
\end{align*}
\] (incur false positive)
big bloom filter

Partition into groups of $L$ bits

$L$ bits

$L$ bits

$L$ bits

$L$ bits

How to decide $L$? (encoded in $2\times \log L$ bits)

- Observation: ensure the num of 1’s in a group is small, e.g., 1
- Suggestion: $L \leq m/n$ ($m$: bloom filter size; $n$: total num of 1’s)

<table>
<thead>
<tr>
<th>$L$</th>
<th>space</th>
<th>false positive ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>$128 (= m/n)$</td>
<td>256MB</td>
<td>3.7%</td>
</tr>
<tr>
<td>$64 (= m/2n)$</td>
<td>512MB</td>
<td>1.5%</td>
</tr>
<tr>
<td>$32 (= m/4n)$</td>
<td>1GB</td>
<td>0.87%</td>
</tr>
<tr>
<td>Base1 (uncompressed)</td>
<td>2GB</td>
<td>1%</td>
</tr>
</tbody>
</table>
Algorithm: PComp
(Position-based compression)

If the bloom filter is sparse enough, can we compress the bloom filter?

00010000001000000000000000000000000000.....00000

64 bits: two 1's

3 9

0 0 0 0 1 1 0 0 1 0 0 1

compressed version

Main idea: Encode the positions of 1’s

Query: is the $i$-th position 1? Scan the position array to find $i$
big bloom filter

Partition into groups of $L$ bits

$L$ bits  $L$ bits  $L$ bits  $L$ bits

How to decide $L$?
- Observation: need to scan positions, do not cross CPU cache lines
- **Encoded positions fit into a cache line (512 bits) after compression**

$L \geq 512 \Rightarrow$ Each position takes $\log L \geq 9$ bits $\Rightarrow$ round up to 16 bits $\Rightarrow$ $L \leq 65536$

A cache line stores at most $512 / 16 = 32$ positions, what if the num of 1’s in $L$ bits exceeds 32? All pass *(incur extra false positive, tune $L$)*
Experiments

**Query set:** 1 billion 8-byte random integers

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Meaning</th>
<th>Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>$n$</td>
<td>data size</td>
<td>100M, 10M, 1M, 100K</td>
</tr>
<tr>
<td>$p$</td>
<td>false positive ratio</td>
<td>1%, 5%</td>
</tr>
<tr>
<td>$k$</td>
<td>num of hash functions</td>
<td>1, 2, 4, 8</td>
</tr>
</tbody>
</table>

**Evaluation metrics**
- Space (*satisfying the given false positive ratio*)
- Time

**Machine**
- CPU: Intel(R) Core(TM) i7-4770 CPU @ 3.40GHz
- Cache: L1: 32KB, L2: 256KB, L3: 8MB
Results on 100M

- Time: Base1 and Cache2 are winners
- Space: Base4 and Cache4
- CacheK is faster than BaseK since cannot fit in cache
- MComp and PComp are worse than Cache4
Results on 10M

- Time: **Base1** and **Cache2** are winners
- Space: **Base4** and **Cache4**
- CacheK is faster than BaseK since cannot fit in cache
- MComp and PComp are worse than Cache4

false positive ratio = 1%
Results on 1M

- Time: **Base2** and **Cache2** are the winners and 3x faster than **Base1**
  - Because Base1 cannot fit into cache while Base2 and Cache2 can
  - Better cache locality
- **BaseK** and **CacheK** have similar performance
  - Because both can fit in cache
Results on 100K

- **Time**: Base1 becomes the winner again
  - Because all can fit in cache
- **CacheK** is slightly worse than BaseK
  - Because CacheK needs to compute which cache line first, then offset

false positive ratio = 1%
Effect of false positive ratio (fpr)

**Space:** 1x ~ 4x smaller when fpr = 5%

**Time:** 3% ~ 21% faster (better cache locality)

Data size: 10 million
Which algorithm to choose? (time)

<table>
<thead>
<tr>
<th>fpr = 1%</th>
<th>fpr = 5%</th>
</tr>
</thead>
<tbody>
<tr>
<td>n = 100M</td>
<td>Base1 Cache2</td>
</tr>
<tr>
<td>n = 10M</td>
<td>Base1 Cache2</td>
</tr>
<tr>
<td>n = 1M</td>
<td></td>
</tr>
<tr>
<td>n = 100K</td>
<td></td>
</tr>
</tbody>
</table>

- All cannot fit in cache
- Some fit in cache
- All fit in cache

speedup over Base1
Which algorithm to choose? (space)

<table>
<thead>
<tr>
<th></th>
<th>All cannot fit in cache</th>
<th>Some fit in cache</th>
<th>All fit in cache</th>
</tr>
</thead>
<tbody>
<tr>
<td>( n = 100M )</td>
<td>Base4 Cache4 (16x)</td>
<td>Base4 Cache4 (8x)</td>
<td>Base4 Cache4 (16x)</td>
</tr>
<tr>
<td>( n = 10M )</td>
<td>Base4 Cache4 (8x)</td>
<td>Base4 Cache4 (8x)</td>
<td>Base4 Cache4 (16x)</td>
</tr>
<tr>
<td>( n = 1M )</td>
<td>Base4 Cache4 (8x)</td>
<td>Base4 Cache4 (4x)</td>
<td>Base4 Cache4 (2x)</td>
</tr>
<tr>
<td>( n = 100K )</td>
<td>Base4 Cache4 (16x)</td>
<td>Base4 Cache4 (4x)</td>
<td>Base4 Cache4 (2x)</td>
</tr>
</tbody>
</table>

fpr = 1%  
fpr = 5%
Conclusion & future work

• No single optimal algorithm, depending on
  – Hardware platform, e.g., cache size
  – Data size
  – False positive ratio

• Future
  – Apply ML techniques to automatically choose the best algorithm based on observed runtimes for the HW platform and problem shape