Reducing Bloom Filter CPU Overhead in LSM-Trees on Modern Storage Devices

Ju Hyoung Mun, Zichen Zhu, Aneesh Raman, Manos Athanassoulis
jmun@bu.edu, zczhu@bu.edu, aneeshr@bu.edu, mathan@bu.edu

presentation at DAMON 2021
Log-Structured Merge Trees

Widely adopted because they balance read performance and ingestion
Log-Structured Merge Trees

buffer

L1
L2
L3
Log-Structured Merge Trees
Log-Structured Merge Trees

.buffer

L1
size ratio = \( T \)

L2

L3

exponentially larger capacity
Log-Structured Merge Trees

buffer

L1
L2
L3

exponentially larger capacity
Log-Structured Merge Trees

buffer

exponentially larger capacity
Log-Structured Merge Trees

get($k$)

buffer

Block Cache

L1

L2

L3

Block Cache
buffer

get$(k)$

Block Cache

$L_1$

$L_2$

$L_3$

Log-Structured Merge Trees

get$(k)$

$F_{1,1}?$

buffer
Log-Structured Merge Trees

buffer

get(k)

Block Cache

L1

F_{1,1}

L2

k

L3
Log-Structured Merge Trees

get($k$)

buffer

Block Cache

$L1$

$L2$

$L3$

$F_{2,2}$?

$F_{1,1}$

$F_{2,2}$

$F_{2,2}$
Log-Structured Merge Trees

get(k)  
buffer  

Block Cache

F₁,₁  F₂,₂

I₂,₂?

L₁

L₂

L₃
Log-Structured Merge Trees

get(k)

buffer

Block Cache

L1

L2

L3

F_{1,1} F_{2,2} I_{2,2} D_{2,2}

k

D_{2,2}

get(k)

k?
Log-Structured Merge Trees

get(x)

buffer

F_{1,1}?

Block Cache

L1

L2

L3

F_{1,1} F_{2,2} I_{2,2} D_{2,2}
Log-Structured Merge Trees

buffer

get(x)

Block Cache

F_{1,1} F_{2,2} I_{2,2} D_{2,2}

L1

L2

L3
Log-Structured Merge Trees

get(x)

buffer

Block Cache

L1
L2
L3
Log-Structured Merge Trees

get(x) → D_{2,2}?

buffer

Block Cache

F_{1,1}  F_{2,2}  I_{2,2}  D_{2,2}

L1

L2

L3
Log-Structured Merge Trees

get(x)

buffer

Block Cache

L1

L2

L3
Memory Pressure in LSM-trees
Memory vs. Storage

The price drop in memory has been slower than storage making it hard to maintain the same memory-to-data ratio

<table>
<thead>
<tr>
<th>Metric</th>
<th>DRAM</th>
<th></th>
<th></th>
<th></th>
<th>HDD</th>
<th></th>
<th></th>
<th></th>
<th>SATAFlash SSD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit price ($)</td>
<td>5k</td>
<td>15k</td>
<td>48</td>
<td>80</td>
<td>30k</td>
<td>2k</td>
<td>80</td>
<td>49</td>
<td>1k</td>
</tr>
<tr>
<td>Unit capacity</td>
<td>1MB</td>
<td>1GB</td>
<td>1GB</td>
<td>16GB</td>
<td>180MB</td>
<td>9GB</td>
<td>250GB</td>
<td>2TB</td>
<td>32GB</td>
</tr>
<tr>
<td>$/MB</td>
<td>5k</td>
<td>14.6</td>
<td><strong>0.05</strong></td>
<td><strong>0.005</strong></td>
<td><strong>0.0003</strong></td>
<td><strong>0.00002</strong></td>
<td><strong>0.03</strong></td>
<td><strong>0.0005</strong></td>
<td></td>
</tr>
<tr>
<td>Random IOPS</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>5</td>
<td>64</td>
<td>83</td>
<td>200</td>
<td>6.2k</td>
</tr>
<tr>
<td>Sequential b/w (MB/s)</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>1</td>
<td>10</td>
<td>300</td>
<td>200</td>
<td>66</td>
</tr>
</tbody>
</table>

The Five-Minute Rule 30 Years Later and Its Impact on the Storage Hierarchy, Communications of the ACM, 2019
Memory Pressure in LSM-trees

For 1TB data, 1.3GB filter & 17.2GB index
11% space amplification,
1KB entry, 64B key, bpk 10

Size of each block increases
Memory pressure
As the available memory decreases, the read bytes per query increase rapidly.
Are all filter blocks equally important?

State-of-the-art LSM designs treat all BFs equally
Access Frequency Patterns

Even in a perfectly uniform workload, **80% of the lookups** are directed to **44~46% of the SST files**.
Bloom Filter

An $m$-bit vector
$n$ elements are stored
$k$ hash indexes

Always access all $k$ indexes for positive queries
Bloom Filter

$m$-bit vector
$n$ elements are stored
$k$ hash indexes

Is the entire filter useful?
Bloom Filter

- $m$-bit vector
- $n$ elements are stored
- $k$ hash indexes

For all $k$ hash indexes, the formula for the number of probes is:

$$probes_{empty} = 1 + \frac{1}{2} + \frac{1}{2^2} + \cdots + \frac{1}{2^{k-1}} = \sum_{d=1}^{k} \frac{1}{2^{k-1}} = 2 - \frac{1}{2^{k-1}}$$
Bloom Filter

A Bloom filter is an m-bit vector that can store up to n elements using k hash indexes. To check if an element x is in the set, k hash functions $h_i(x)$ are computed, and if all bits corresponding to these hash indexes are set, then x is assumed to be in the set. Otherwise, x is assumed to be not in the set. If the element is not in the set, the probability of falsely reporting its presence (false positive) is approximately:

$$\text{probes}_{\text{empty}} = 1 + \frac{1}{2} + \frac{1}{2^2} + \cdots + \frac{1}{2^{k-1}} = \sum_{d=1}^{k} \frac{1}{2^{k-1}} = 2 - \frac{1}{2^{k-1}}$$
Modular Bloom Filter

$m$-bit vector
$n$ elements are stored
$k$ hash indexes
d modules

An MBF is a collection of $D$ Bloom filters
- $m_d$-bit vector
- $n$ elements
- $k_d$ hash indexes
Modular Bloom Filter

- $m$-bit vector
- $n$ elements are stored
- $k$ hash indexes
- $d$ modules

Example:

- $x$?
- Module #3: Negative

Hash functions:

$$h_1(x)$$

$$h_2(x)$$
Modular Bloom Filter

- $m$-bit vector
- $n$ elements are stored
- $k$ hash indexes
- $d$ modules

$y$? positive

$h_1(y)$  
$h_2(y)$  
$h_3(y)$

module #1  
module #2  
module #3
Modular Bloom Filter

False positive rate

*FPR close-to-theoretical*

Avg. number of module accesses
Modular Bloom Filter

False positive rate

*FPR close-to-theoretical*

Avg. # of module accesses

vs.

Avg. size accessed

*Less space requirement*
Modular Bloom Filter

- \( m \)-bit vector
- \( n \) elements are stored
- \( k \) hash indexes
- \( d \) modules

\[ y? \text{ positive} \]

- \( h_1(y) \)
- \( h_2(y) \)
- \( h_3(y) \)

**MBFs are not useful for non-empty queries.**

*What if we know the queries would be non-empty in advance?*
Skipping Modules

**Utility**: a measure of the benefit of a filter or a module

\[ u_{l,i,d} = \exp IO_{l,i,d} - \exp IO_{l,i,d-1} \]

The expected number of I/Os that can be reduced by using \(d\)-th module

**Expected number of I/Os**

\[ \exp IO_{l,i,d} = \beta_{l,i} \cdot \left( \alpha_{l,i} + (1 - \alpha_{l,i}) \cdot f_{sm}^d \right) \]

- \( \beta_{l,i} \): access frequency
- \( \alpha_{l,i} \): non-empty queries
- \( f_{sm} \): FPR of a single module
- \( f_{sm} \): false positives from empty queries
- \( l \)-th level
- \( i \)-th SST file
- \( d \)-th module

**Utility is high if file is frequently accessed, or queries are empty**
Skipping Modules

**Skipping Modules** based on their utilities

\[ u_{l,i,d} = \expIO(l,i,d) - \expIO(l,i,d-1) \]  // calc module’s utility

**if** \[ u_{l,i,d} < \text{threshold}_d \] **then**  // skip module when there’s no benefit
  return \text{true}

**else**  // otherwise, keep querying modules
  result = QueryModule( key, module_{l,i,d} )
Modular Bloom filter & Skipping Algorithm & Sharing Hashing + LSM-tree

Sharing Hashing with Modular Bloom filters (SHaMBa)
Experimental Evaluation
# Experiment Settings

## LSM-tree tuning

<table>
<thead>
<tr>
<th>Term</th>
<th>Value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>E</td>
<td>64</td>
<td>entry size (B)</td>
</tr>
<tr>
<td>K</td>
<td>32</td>
<td>key size (B)</td>
</tr>
<tr>
<td>B</td>
<td>64</td>
<td>block size (#entries)</td>
</tr>
<tr>
<td>P</td>
<td>1024</td>
<td>buffer size/file size (#blocks)</td>
</tr>
<tr>
<td>T</td>
<td>4</td>
<td>size ratio</td>
</tr>
<tr>
<td>b</td>
<td>10</td>
<td>bits per key for filters</td>
</tr>
</tbody>
</table>

## Size of blocks

<table>
<thead>
<tr>
<th>Term</th>
<th>Value</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$S_D$</td>
<td>4</td>
<td>data block size (KB)</td>
</tr>
<tr>
<td>$S_I$</td>
<td>32</td>
<td>index block size (KB)</td>
</tr>
<tr>
<td>$S_F$</td>
<td>80</td>
<td>filter block size (KB)</td>
</tr>
</tbody>
</table>
## Approaches Tested

<table>
<thead>
<tr>
<th>Tuning knobs of SHaMBa</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Term</td>
<td>Value</td>
</tr>
<tr>
<td>number of modules</td>
<td>1, 2, 3, or 7</td>
</tr>
<tr>
<td>Size of each module</td>
<td>equal or proportional</td>
</tr>
<tr>
<td>skipping algorithm</td>
<td>none, partial (P), or full (F)</td>
</tr>
</tbody>
</table>

### Approaches Tested

- `state-of-the-art`
- `SHaMBa-eq`
- `SHaMBa-eq-P`
- `SHaMBa-eq-F`
- `SHaMBa-prop`
- `SHaMBa-prop-P`
- `SHaMBa-prop-F`
Impact of number of modules

Workload: Uniform, Entry size: 64B, #Entries: 30K
Tuning: no skipping algorithm, equal sized modules

SHaMBa enhances the lookup performance for empty queries
Impact of number of modules

**Workload:** Uniform, Entry size: 64B, #Entries: 30K

**Tuning:** no skipping algorithm, equal sized modules

SHaMBa Performs Best with Smaller Modules
Impact of number of modules

Smaller modules are more beneficial
Impact of number of modules

**Workload:** Uniform, Entry size: 64B, Entries: 30K
**Tuning:** full skipping algorithm, equal sized modules

Skipping modules reduces the impact of the number of the modules
Impact of number of modules

Workload: Uniform, Entry size: 64B, #Entries: 30K
Tuning: full skipping algorithm, equal sized modules

Skipping modules reduces the impact of the number of the modules
**SHaMBa with Partitioned Index/Filter**

**Workload:** Uniform, Entry size: 64B, #Entries: 30K

**Tuning:** 2 equal sized modules

---

- partitioned
- partitioned + SHaMBa-eq
- partitioned + SHaMBa-eq-P
- partitioned + SHaMBa-eq-F

---

**SHaMBa boosts partitioned index/filter under severe memory pressure**
SHaMBa with Monkey

Monkey allocates more bits per element in the shallower levels to aggressively reduce their false positives

Monkey: Optimal Navigable Key-Value Store, ACM SIGMOD 2017

Workload: Uniform, Entry size: 64B, #Entries: 30K
Tuning: 2 equal sized modules

SHaMBa further improves performance of Monkey
SHaMBA-eq with RocksDB

Workload: Uniform, Entry size: 64B, #Entries: 30K
Tuning: 2 equal sized modules, RocksDB version 6.19.3

SHaMBA-eq accelerates point lookups
SHaMBa-prop with RocksDB

Workload: Uniform, Entry size: 64B, #Entries: 30K
Tuning: 2 proportional sized modules, RocksDB version 6.19.3
SHaMBA-prop with RocksDB

**Workload:** Uniform, Entry size: 64B, #Entries: 30K

**Tuning:** 2 proportional sized modules, RocksDB version 6.19.3

SHaMBA-prop accelerates point lookups
SHaMBa on various Devices

Workload: Uniform, Entry size: 64B, #Entries: 30K
Tuning: 2 equal sized modules, RocksDB version 6.19.3

SHaMBa also benefits faster storage devices
SHaMBa with larger index

Workload: Uniform (all empty), Entry size: 128B, #Entries: 30K
Tuning: 2 equal sized modules, RocksDB version 6.19.3

SHaMBa performs best when filters are larger than indexes
Conclusion

- Modular Bloom filters (MBFs)
  - a BF variant that consists of multiple module
  - enable smooth navigation of the memory vs. performance trade-off

- SHaMBa
  - a novel LSM-based key-value engine
  - specifically addresses performance loss due to memory pressure
  - the same average number of I/Os, with 1/3 of the memory by the state of the art

Thank you!