LSM-Trees Under (Memory) Pressure

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

presentation at ADMS 2022
Log-Structured Merge Trees

Widely adopted because they balance read performance and ingestion

- RocksDB
- DynamoDB
- HBase
- WT
- levelDB
- cassandra
Log-Structured Merge Trees

buffer

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

buffer

L1
L2
L3

size ratio = T

exponentially larger capacity
Log-Structured Merge Trees

Buffer

L1

L2

L3

Organized in SST files
Log-Structured Merge Trees

Buffer

Filter and index blocks to enhance the lookup performance
Log-Structured Merge Trees

get\(k\)

buffer

L1

L2

L3

k
Log-Structured Merge Trees

get($k$)

buffer

Block Cache

L1

L2

L3

get($k$)
Log-Structured Merge Trees

get(k)

buffer

F_{1,1}?

Block Cache

L1

L2

L3
Log-Structured Merge Trees

get(k)

buffer

Block Cache

L1

L2

L3

F_{1,1}?
Log-Structured Merge Trees

get($k$)

buffer

Block Cache

$F_{1,1}$

$L1$

$L2$

$L3$

$k$
Log-Structured Merge Trees

get(k) → Buffer → Block Cache → L1 → L2 → L3

$F_{1,1}$
Log-Structured Merge Trees

get($k$)
Log-Structured Merge Trees

get\( (k) \)

buffer

Block Cache

\[ F_{1,1} \]

\[ F_{2,2} \]

L1

L2

L3

k
Log-Structured Merge Trees

get(k)

buffer

F_{2,2}?

Block Cache

F_{1,1}

L1

L2

k

L3
Log-Structured Merge Trees

get(k)  F_{2,2}?  Block Cache
buffer → F_{1,1} → L1 → L2 → L3
Log-Structured Merge Trees

get\( (k) \)

buffer

Block Cache

\( F_{1,1} \)

\( F_{2,2} \)

L1

L2

L3

k
Log-Structured Merge Trees

get($k$)

buffer

Block Cache

F$_{1,1}$ F$_{2,2}$

L1

L2

L3

$F_{1,1}$ $F_{2,2}$
Log-Structured Merge Trees

get(k)

buffer

Block Cache

F\textsubscript{1,1} F\textsubscript{2,2}

L1

L2

L3

k
Log-Structured Merge Trees

get\( (k) \)

buffer

Block Cache

L1

L2

L3

\( I_{2,2} ? \)
Log-Structured Merge Trees

get($k$)

buffer

Block Cache

L1

L2

L3

$F_{1,1}$  $F_{2,2}$

$k$

I$_{2,2}$?
Log-Structured Merge Trees

```
get(k)

buffer

Block Cache

F_1,1  F_2,2

I_2,2?

L1

L2

L3
```

I_2,2
Log-Structured Merge Trees

get($k$)

buffer

Block Cache

$F_{1,1}$ $F_{2,2}$ $L_{2,2}$

L1

L2

L3

$k$
Log-Structured Merge Trees

get(k)

buffer

Block Cache

L1

L2

L3

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

D_{2,2,1}

k
Log-Structured Merge Trees

buffer

get\((k)\)

Block Cache

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

L1

L2

L3

k
Log-Structured Merge Trees

get($k$)

buffer

D$_{2,2,1}$?

Block Cache

F$_{1,1}$  F$_{2,2}$  I$_{2,2}$

L1

L2

L3

k
Log-Structured Merge Trees

get\( (k) \)

buffer

\[ D_{2,2,1} \]

Block Cache

\[ F_{1,1}, F_{2,2}, I_{2,2} \]

L1

L2

L3

get\( (k) \)

buffer

\[ D_{2,2,1} \]

Block Cache

\[ F_{1,1}, F_{2,2}, I_{2,2} \]

L1

L2

L3

get\( (k) \)

buffer

\[ D_{2,2,1} \]

Block Cache

\[ F_{1,1}, F_{2,2}, I_{2,2} \]

L1

L2

L3
Log-Structured Merge Trees

buffer

get(k)

D_{2,2,1}? Block Cache

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

L1

L2

L3

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

buffer

get($k$)

Block Cache

L1

L2

L3

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

get\(k\)

buffer

Block Cache

\(F_{1,1}, F_{2,2}, I_{2,2}, D_{2,2,1}\)

L1

L2

L3

k
Log-Structured Merge Trees

get\(k\) → buffer

Block Cache: \(F_{1,1}, F_{2,2}, I_{2,2}, D_{2,2,1}\)

L1
L2: \(k\)
L3
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_{2,2,1}

F_{1,1}, x

...
Log-Structured Merge Trees

get(x) -> Buffer

Block Cache: \( F_{1,1} \), \( F_{2,2} \), \( I_{2,2} \), \( D_{2,2,1} \)

L1
L2
L3

34
Log-Structured Merge Trees

`get(x)`

Buffer

Block Cache

L1

L2

L3
Log-Structured Merge Trees

```
get(x)
buffer  F_{2,2}?
```

Block Cache

```
F_{1,1}  F_{2,2}  I_{2,2}  D_{2,2,1}
```
get($x$) \\

buffer

Block Cache

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

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

get(x)

buffer

Block Cache

L1

L2

L3
Log-Structured Merge Trees

get(x)

buffer

Block Cache

L1

L2

L3

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

I_{2,2}?
Log-Structured Merge Trees

Buffer

get(x)

Block Cache

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

L1

L2

L3
Log-Structured Merge Trees

get(x)

buffer

I_{2,2}? Block Cache

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

D_{2,2,2}

L1

L2

L3
Log-Structured Merge Trees

buffer

get(x)

D_{2,2,2}?

Block Cache

\[
\begin{array}{c}
F_{1,1} & F_{2,2} & I_{2,2} & D_{2,2,1} \\
\times & & \checkmark & \checkmark \\
\end{array}
\]

L1

L2

L3

x
Log-Structured Merge Trees

get(x)

buffer

D_{2,2,2}?

Block Cache

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

L1

L2

L3
Log-Structured Merge Trees

get(x)

buffer

Block Cache

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

L1

L2

L3

D_{2,2,2}
Log-Structured Merge Trees

get(x)

Block Cache

buffer

L1

L2

L3
Log-Structured Merge Trees

get(x)

buffer

Block Cache

L1

L2

L3
Log-Structured Merge Trees

get(\(x\))

buffer

\(x?\)

Block Cache

L1

L2

L3

get(\(x\))
Log-Structured Merge Trees

buffer

get(x)

Block Cache

L1

L2

L3

get(x)
Can we always keep useful block in block cache?
Memory Pressure in LSM-trees
# Memory vs. Storage

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

<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>
<th>2007</th>
<th>2018</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit price ($)</td>
<td>5k</td>
<td>15k</td>
<td>48</td>
<td>80</td>
<td>30k</td>
<td>2k</td>
<td>80</td>
<td>49</td>
<td>1k</td>
<td>415</td>
<td></td>
</tr>
<tr>
<td>Unit capacity</td>
<td>1MB</td>
<td>1GB</td>
<td>1GB</td>
<td>16GB</td>
<td>180MB</td>
<td>9GB</td>
<td>250GB</td>
<td>2TB</td>
<td>32GB</td>
<td>800GB</td>
<td></td>
</tr>
<tr>
<td>$/MB</td>
<td>5k</td>
<td>14.6</td>
<td>0.05</td>
<td>0.005</td>
<td>83.33</td>
<td>0.22</td>
<td>0.0003</td>
<td>0.00002</td>
<td>0.03</td>
<td>0.0005</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>
<td>67k (r)/20k (w)</td>
<td></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>
<td>500 (r)/460 (w)</td>
<td></td>
</tr>
</tbody>
</table>
## Memory vs. Storage

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

<table>
<thead>
<tr>
<th>Metric</th>
<th>DRAM</th>
<th>HDD</th>
<th>SATAFlash SSD</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit price ($)</td>
<td>5k</td>
<td>15k</td>
<td>48</td>
</tr>
<tr>
<td>Unit capacity</td>
<td>1MB</td>
<td>1GB</td>
<td>1GB</td>
</tr>
<tr>
<td>$/MB</td>
<td>5k</td>
<td>14.6</td>
<td><strong>0.05</strong></td>
</tr>
<tr>
<td>Random IOPS</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Sequential b/w (MB/s)</td>
<td>–</td>
<td>–</td>
<td>–</td>
</tr>
</tbody>
</table>
# Memory vs. Storage

The price drop in memory has been slower than storage

<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>
<th></th>
<th>2007</th>
<th>2018</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit price ($)</td>
<td>5k</td>
<td>15k</td>
<td>48</td>
<td>80</td>
<td>30k</td>
<td>2k</td>
<td>80</td>
<td>49</td>
<td>1k</td>
<td>415</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unit capacity</td>
<td>1MB</td>
<td>1GB</td>
<td>1GB</td>
<td>16GB</td>
<td>180MB</td>
<td>9GB</td>
<td>250GB</td>
<td>2TB</td>
<td>32GB</td>
<td>800GB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$/MB</td>
<td>5k</td>
<td>14.6</td>
<td>0.05</td>
<td>0.005</td>
<td>83.33</td>
<td>0.22</td>
<td>0.0003</td>
<td>0.00002</td>
<td>0.03</td>
<td>0.0005</td>
<td></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>
<td>67k (r)/20k (w)</td>
<td></td>
<td></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>
<td>500 (r)/460 (w)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The Five-Minute Rule 30 Years Later and Its Impact on the Storage Hierarchy, Communications of the ACM, 2019
Memory vs. Storage

<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>83.33</td>
<td>0.22</td>
<td><strong>0.0003</strong></td>
<td><strong>0.00002</strong></td>
<td><strong>0.03</strong></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

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

Data size ↑
Memory Pressure in LSM-trees

For 1TB data,
1.3GB filter & 17.2GB index
11% space amplification,
1KB entry, 64B key, bpk 10
Memory Pressure in LSM-trees

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

![Graph showing the relationship between memory budget and read bytes/lookup (KB). The graph indicates a decrease in read bytes as the memory budget increases.]
Lookup cost under memory pressure

![Graph showing the relationship between read bytes/lookup (KB) and memory budget (%). The graph shows a downward trend with memory pressure, indicating a decrease in read bytes/lookup as the memory budget increases. Key points on the graph include:

- At 10% memory budget, the read bytes/lookup is approximately 250 KB.
- At 40% memory budget, the read bytes/lookup is approximately 7.3x lower than at 10%.
- At 70% memory budget, the read bytes/lookup is approximately 1.4x lower than at 40%.
- At 100% memory budget, the read bytes/lookup is approximately 1x lower than at 70%.

The graph visually demonstrates the efficiency gain in read operations as the memory budget increases under memory pressure.]
Lookup cost under 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

![CDF graph with different access frequency patterns]

- **Uniform**
- **Normal (low skew)**
- **Normal (high skew)**
- **YCSB (Uniform)**

# SST files (descending order of access frequency)
Access Frequency Patterns

![Access Frequency Patterns Graph]

- **CDF**
- **# SST files (descending order of access frequency)**

- **Uniform**
- **Normal (low skew)**
- **Normal (high skew)**
- **YCSB (Uniform)**
Even in a perfectly uniform workload, 80% of the lookups are directed to 44~46% of the SST files.
Bloom Filter

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

$y$?
Bloom Filter

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

$h_1(y)$

$y$?
Bloom Filter

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

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

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

$m$-bit vector
Bloom Filter

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

$y$?

positive

$h_1(y)$

$h_2(y)$

$h_3(y)$

$m$-bit vector
Bloom Filter

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

\[ h_i(y) \]

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

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

$x$?
Bloom Filter

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

\[ x \]

\[ h_1(x) \]
Bloom Filter

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

$x$?

$h_1(x)$

$h_2(x)$

$m$-bit vector
Bloom Filter

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

$x$? negative
Bloom Filter

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

$x$? negative

Is the entire filter useful?
Bloom Filter

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

\[ h_1(x) \]
\[ h_2(x) \]

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

An m-bit vector contains n elements, and k hash indexes are used. The probability of a successful probe is calculated as follows:

\[ \text{probes}_{\text{empty}} = 1 + \frac{1}{2} + \frac{1}{2^2} + \ldots + \frac{1}{2^{k-1}} = \sum_{d=1}^{k} \frac{1}{2^{k-1}} = 2 - \frac{1}{2^{k-1}} \]
Bloom Filter

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

$h_1(x)$
$h_2(x)$

$x$? negative

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

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

$x$?

negative

\[
\text{probes}_{\text{empty}} = 1 + \frac{1}{2} + \frac{1}{2^2} + \ldots + \frac{1}{2^{k-1}} = \sum_{d=1}^{k} \frac{1}{2^{k-1}} = 2 - \frac{1}{2^{k-1}}
\]
Bloom Filter

$m$-bit vector

$n$ elements are stored

$k$ hash indexes

$x? \quad \text{negative}$

\[
\begin{align*}
\text{probes}_{\text{empty}} &= 1 + \frac{1}{2} + \frac{1}{2^2} + \ldots + \frac{1}{2^{k-1}} \\
&= \sum_{d=1}^{k} \frac{1}{2^{k-1}} = 2 - \frac{1}{2^{k-1}}
\end{align*}
\]
Bloom Filter

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

$h_1(x)$
$h_2(x)$

$x_? \text{ negative}

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}}$

for all $k$ hash indexes
Bloom Filter

*m-bit vector*

*n elements are stored*

*k hash indexes*

\[ h_1(x) \]

\[ h_2(x) \]

\[ x? \text{ negative} \]

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

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

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

An MBF is a collection of $D$ Bloom filters
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 store
- $k$ hash indexes
- $d$ modules

$x$?
Modular Bloom Filter

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

$h_1(x)$

$x$?
Modular Bloom Filter

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

$h_1(x)$
$h_2(x)$

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

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

$x$? negative

$h_1(x)$
$h_2(x)$

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

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

$y$?

$h_1(y)$

$h_2(y)$

$h_3(y)$

Module #1

Module #2

Module #3
Modular Bloom Filter

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

For $y$: positive

$h_1(y)$

$h_2(y)$

$h_3(y)$

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

False positive rate

![Diagram showing the false positive rate for different numbers of modules. The graph indicates that the false positive rate remains relatively constant across various numbers of modules.]
Modular Bloom Filter

False positive rate

FPR close-to-theoretical
Modular Bloom Filter

False positive rate

*FPR close-to-theoretical*

Avg. # of module accesses
Modular Bloom Filter

False positive rate

**FPR close-to-theoretical**

Avg. # of module accesses vs. Avg. size accessed
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

$y$?

Positive

$h_1(y)$

$h_2(y)$

$h_3(y)$
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

MBFs are not useful for positive queries.
Modular Bloom Filter

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

$y$? positive

$h_1(y)$

$h_2(y)$

$h_3(y)$

MBFs are not useful for positive queries.

What if we know something more about the queries?
Skipping Modules

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

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

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

\[ u_{l,i,d} = \text{exp}I/O_{l,i,d} - \text{exp}I/O_{l,i,d-1} \]

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

**Expected number of I/Os**

\[ \text{exp}I/O_{l,i,d} = \beta_{l,i} \cdot (\alpha_{l,i} + (1 - \alpha_{l,i}) \cdot f_{sm}^d) \]

- \(l\)-th level
- \(i\)-th SST file
- \(d\)-th module
- \(f_{sm}\): FPR of a single module
Skipping Modules

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

\[ u_{l,i,d} = \exp I/O_{l,i,d} - \exp I/O_{l,i,d-1} \]

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

**Expected number of I/Os**

\[ \exp I/O_{l,i,d} = \beta_{l,i} \cdot (\alpha_{l,i} + (1 - \alpha_{l,i}) \cdot f_{sm}^{d}) \]

\( l \)-th level
\( i \)-th SST file
\( d \)-th module

\( f_{sm} \): FPR of a single module
Skipping Modules

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

\[ u_{l,i,d} = \exp I/O_{l,i,d} - \exp I/O_{l,i,d-1} \]

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

**Expected number of I/Os**

\[ \exp I/O_{l,i,d} = \beta_{l,i} \cdot (\alpha_{l,i} + (1 - \alpha_{l,i}) \cdot f_s^d) \]

- \( l \)-th level
- \( i \)-th SST file
- \( d \)-th module
- \( f_s^d \): FPR of a single module
- non-empty queries
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*

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

- \( l \)-th level
- \( i \)-th SST file
- \( d \)-th module
- \( f_{sm} \): FPR of a single module
- non-empty queries
Skipping Modules

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

\[ u_{l,i,d} = \exp I/O_{l,i,d} - \exp I/O_{l,i,d-1} \]

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

**Expected number of I/Os**

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

- \( \beta_{l,i} \): FPR of a single module on non-empty queries
- \( \alpha_{l,i} \): FPR of a single module on empty queries
- \( f_{sm}^d \): FPR of a single module
- \( l \)-th level
- \( i \)-th SST file
- \( d \)-th module
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 the \( d \)-th module

*Expected number of I/Os*

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

- \( \beta_{l,i} \) : FPR of a single module
- \( \alpha_{l,i} \) : FPR of a non-empty query
- \( f_{sm}^d \) : FPR of a module
- \( l \)-th level
- \( i \)-th SST file
- \( d \)-th module
- \( f_{sm} \) : FPR of a single module
**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**

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

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

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

\[
\mathcal{U}_l,i,d = \exp IO_{l,i,d} - \exp IO_{l,i,d-1}
\]

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

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

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

*Skipping Modules* based on their utilities
Skipping Modules

**Skipping Modules** based on their utilities

\[ u_{l,i,d} = \expIO(l,i,d) - \expIO(l,i,d-1) \]

if \( u_{l,i,d} < \text{threshold}_d \) then
  return true

else
  result = QueryModule( key, module_{l,i,d} )
Skipping Modules

Skipping Modules based on their utilities

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

if \( u_{l,i,d} < \text{threshold}_d \) then
   return true

else
   result = QueryModule( key, module_{l,i,d} )
Skipping Modules based on their utilities

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

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

\[
\text{else} \\
\text{result} = \text{QueryModule( key, module}_{l,i,d} )
\]
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 true

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

<table>
<thead>
<tr>
<th>Tuning knobs of SHaMBa</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<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**
  - \( \text{SHaMBa-eq} \)
  - \( \text{SHaMBa-eq-P} \)
  - \( \text{SHaMBa-eq-F} \)
- **SHaMBa-prop**
  - \( \text{SHaMBa-prop-P} \)
  - \( \text{SHaMBa-prop-F} \)
Approaches Tested

<table>
<thead>
<tr>
<th>Tuning knobs of SHaMBa</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<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 <strong>full ($F$)</strong></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-$F$**
Approaches Tested

<table>
<thead>
<tr>
<th>Term</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<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>

- state-of-the-art
- SHaMBa-eq
- SHaMBa-eq-P
- SHaMBa-eq-Ф
- SHaMBa-prop
- SHaMBa-prop-P
- SHaMBa-prop-Ф
Approaches Tested

Tuning knobs of SHaMBa

<table>
<thead>
<tr>
<th>Term</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<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
# Approaches Tested

## Tuning knobs of SHaMBa

<table>
<thead>
<tr>
<th>Term</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<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

---

[Graphs showing the impact of memory budget on I/O per lookup for different numbers of modules.]

- **State-of-art**
- 2 modules
- 3 modules
- 7 modules

---

**Memory budget (%):**
- 0
- 20
- 40
- 60
- 80
- 100
- 120
- 140
- 160
- 180

**I/O per lookup:**
- 0
- 10
- 20
- 30
- 40
- 50
- 60
- 70
- 80
- 90

**Cases:**
- All empty
- Half empty
- All non-empty
Impact of number of modules

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

- state-of-art
- 2 modules
- 3 modules
- 7 modules
Impact of number of modules

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

- state-of-art
- 2 modules
- 3 modules
- 7 modules

I/O per lookup vs Memory budget (%)

- all empty
- half empty
- all non-empty
Impact of number of modules

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

![Graph showing impact of number of modules on I/O per lookup with different memory budgets.](image)
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

- blue line: state-of-art
- x-x-x line: 2 modules
- -o-o line: 3 modules
- -o-o line: 7 modules

Graphs show the impact of memory budget (%) on I/O per lookup for different module configurations:
- all empty
- half empty
- all non-empty
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

% of empty queries terminated

Fraction of MBF accessed

0 1

1 modules
Impact of number of modules

![Graph showing the impact of number of modules on the fraction of MBF accessed. The graph compares 1 module (solid line) and 2 modules (dotted line). The x-axis represents the fraction of MBF accessed, ranging from 0 to 1. The y-axis represents the percentage of empty queries terminated, ranging from 0% to 100%. The graph shows that as the fraction of MBF accessed increases, the percentage of empty queries terminated also increases. For 1 module, the percentage remains close to 100% across all fractions of MBF accessed, while for 2 modules, the percentage starts at around 60% and increases as the fraction of MBF accessed increases.](image-url)
Impact of number of modules
Impact of number of modules

![Graph showing the impact of number of modules on the percentage of empty queries terminated.](image)
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

- state-of-art
- 2 modules
- 3 modules
- 7 modules

<table>
<thead>
<tr>
<th>Memory budget (%)</th>
<th>I/O per lookup</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>40</td>
</tr>
<tr>
<td>40</td>
<td>20</td>
</tr>
<tr>
<td>70</td>
<td>10</td>
</tr>
<tr>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>150</td>
<td>0</td>
</tr>
</tbody>
</table>

- all empty
- half empty
- all non-empty
Impact of number of modules

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

![Graphs showing I/O per lookup vs. Memory budget for different numbers of modules: all empty, half empty, all non-empty.](image_url)
Impact of number of modules

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

- state-of-art
- 2 modules
- 3 modules
- 7 modules

---

Impact of number of modules

<table>
<thead>
<tr>
<th>Memory budget (%)</th>
<th>I/O per lookup</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>all empty</td>
</tr>
<tr>
<td></td>
<td>half empty</td>
</tr>
<tr>
<td></td>
<td>all non-empty</td>
</tr>
</tbody>
</table>

144
Workload: Uniform, Entry size: 64B, #Entries: 30K
Tuning: full skipping algorithm, equal sized modules

Impact of number of 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.
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

all empty
half empty
all non-empty
SHaMBA with Partitioned Index/Filter

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

- partitioned
dashed: partitioned + SHaMBA-eq
dotted: partitioned + SHaMBA-eq-P
dash-dotted: partitioned + SHaMBA-eq-F
SHaMBa with Partitioned Index/Filter

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

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 2022

Workload: Uniform, Entry size: 64B, #Entries: 30K
Tuning: 2 equal sized modules
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 2022

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

**Tuning:** 2 equal sized modules
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 2022

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 with RocksDB**

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

Tuning: 2 equal sized modules, RocksDB version 6.19.3

---

**Graphs:**

- **RocksDB**
- **SHaMBa-eq**
- **SHaMBa-eq-P**
- **SHaMBa-eq-F**

---

**Latency (ms) vs Memory budget (%):**

- **All empty**
- **Half empty**
- **All non-empty**

---

155
**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

---

![Graphs showing latency vs. memory budget for different SHaMBa-prop configurations.](image)

- **RocksDB**
- **SHaMBa-prop**
- **SHaMBa-prop-P**
- **SHaMBa-prop-F**

**Legend:**
- All empty
- Half empty
- All non-empty

**Notation:**
- 10 : 90
**Workload**: Uniform, Entry size: 64B, #Entries: 30K

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

**SHaMBa-prop with RocksDB**

- **RocksDB**
- **SHaMBa-prop**
- **SHaMBa-prop-P**
- **SHaMBa-prop-F**
SHaMBa-prop with RocksDB

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

![Graph showing latency vs memory budget for SHaMBa-prop with RocksDB](image)
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 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

---

**Memory Budget**
- 10%
- 40%
- 70%
- 100%
- 150%

**Key size (Index size ÷ Filter size):**
- 8B (0.2×)
- 32B (0.8×)
- 64B (1.6×)
- 96B (2.4×)
- 124B (3.1×)

**Speedup:**
- Larger filter
- Larger index

---

165
SHaMBa with larger index

**Workload:** Uniform (all empty), **Entry size:** 128B, **#Entries:** 30K

**Tuning:** 2 equal sized modules, RocksDB version 6.19.3
SHaMBa with larger index

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

**Workload:** Uniform (all empty), **Entry size:** 128B, **#Entries:** 30K

**Tuning:** 2 equal sized modules, RocksDB version 6.19.3
SHaMBa with larger index

**Workload:** Uniform (all empty), **Entry size:** 128B, Entries: 30K

**Tuning:** 2 equal sized modules, RocksDB version 6.19.3
**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
Conclusion

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