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

Zichen Zhu, Ju Hyoung Mun, Aneesh Raman, Manos Athanassouli
zczhu@bu.edu, jmun@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
Where does the time go?

![Latency/lookup (μs) Diagram]

- **SSD**
  - BF: 2
  - Data: 5
  - Other: 1

- **PCIe SSD**
  - BF: 2
  - Data: 4
  - Other: 1

- **NVM**
  - BF: 3
  - Data: 1
  - Other: 1
Where does the time go?
Where does the time go?

The time spent on Bloom filters dominates for faster storage.
Log-Structured Merge Trees

buffer
Log-Structured Merge Trees

buffer

L1
L2
L3
Log-Structured Merge Trees

buffer

L1

L2

L3

size ratio = T

exponentially larger capacity
Log-Structured Merge Trees

- Buffer
- Bloom filters
- Fence pointers
- L1
- L2
- L3
- Exponentially larger capacity
- Size ratio = T
Log-Structured Merge Trees

get(k)

buffer

Bloom filters

fence pointers

L1

L2

L3

k
Log-Structured Merge Trees

get($k$)

buffer

Bloom filters

fence pointers

L1

L2

L3

$k$
Log-Structured Merge Trees

get($k$)

buffer

Bloom filters

fence pointers

L1

L2

L3

$k$
Log-Structured Merge Trees

get(k)

buffer

Bloom filters

fence pointers

L1

L2

L3

k
Log-Structured Merge Trees

get(k)

buffer

Bloom filters

fence pointers

L1

L2

L3

k

All queries use a BF!
Log-Structured Merge Trees

get(k)

buffer

Bloom filters

fence pointers

L1

L2

L3

k

All queries use a BF!

What is the cost of querying a Bloom filter?
Bloom Filter Query Cost

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

$x$?

$h_1(x)$
$h_2(x)$
$h_3(x)$

the fraction of empty queries

$$k \cdot T_H + T_P + \alpha \cdot f_p \cdot T_D + (1 - \alpha) \cdot T_D$$

Hashing time  Probing time  Data access time
Bloom Filter Query Cost

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

\[ k \cdot T_H + T_P + \alpha \cdot f_p \cdot T_D + (1 - \alpha) \cdot T_D \]
Bloom Filter Query Cost

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

\[ k \cdot T_H + T_P + \alpha \cdot f_p \cdot T_D + (1 - \alpha) \cdot T_D \]

the fraction of empty queries
Bloom Filter Query Cost

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

The fraction of empty queries is given by:

$$k \cdot T_H + T_P + \alpha \cdot f_p \cdot T_D + (1 - \alpha) \cdot T_D$$

- $h_1(x)$
- $h_2(x)$
- $h_3(x)$

The fraction of empty queries.
Bloom Filter Query Cost

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

\[
\begin{align*}
k \cdot T_H &+ T_P + \alpha \cdot f_p \cdot T_D + (1 - \alpha) \cdot T_D \\
\end{align*}
\]

- empty
- non-empty
Bloom Filter Query Cost

\[ k \cdot T_H + T_P + \alpha \cdot f_p \cdot T_D + (1 - \alpha) \cdot T_D \]

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

Diagram:
- \( x \)?
- \( h_1(x) \)
- \( h_2(x) \)
- \( h_3(x) \)
Bloom Filter Query Cost

\[ k \cdot T_H + T_p + \alpha \cdot f_p \cdot T_D + (1 - \alpha) \cdot T_D \]

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

A single hash function, followed by much cheaper bitwise operations.
Bloom Filter Query Cost

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

Given hash function, $h$

$g_i(x) = h(x) + i \cdot \delta$

$\delta = h(x) \ll 17 \lor h(x) \gg 15$

$k \cdot T_H + T_P + \alpha \cdot f_p \cdot T_D + (1 - \alpha) \cdot T_D$
Bloom Filter False Positive Rate

$k$ vs. single hash function
$k$ vs. single hash function

FPR close-to-theoretical
Bloom Filter Lookup Latency

$k$ vs. single hash function
Bloom Filter Lookup Latency

$k$ vs. single hash function

Latency/lookup (μs)

Key Size (bytes)

8 16 32 64 128 256 512

All $k$  MM64 HS

$k$ times
What is the Lookup Cost in LSM-Trees?
What is the Lookup Cost in LSM-Trees?

Leveling  
read-optimized

Tiering  
write-optimized
Leveling
read-optimized

1 run per level

Tiering
write-optimized

$T$ runs per level
Lookup Cost in a Leveled LSM-Tree

1 run per level

Lookup cost in level $i$, $\mathcal{T}(i)$
Lookup Cost in a Leveled LSM-Tree

1 run per level

Lookup cost in level $i$, $\mathcal{T}(i)$

- empty ($\alpha_i$)
  \[ \alpha_i \cdot (T_H + T_P + f_p \cdot T_D) \]

- non-empty ($1 - \alpha_i$)
  \[ (1 - \alpha_i) \cdot (T_H + T_P + T_D) \]
Lookup Cost in a Leveled LSM-Tree

1 run per level

Lookup cost in level $i$, $\mathcal{J}(i)$

- empty ($\alpha_i$)
  $$\alpha_i \cdot (T_H + T_P + f_p \cdot T_D)$$

- non-empty ($1 - \alpha_i$)
  $$(1 - \alpha_i) \cdot (T_H + T_P + T_D)$$

$$\mathcal{J}(i) = T_H + T_P + \alpha_i \cdot f_p \cdot T_D + (1 - \alpha_i) \cdot T_D$$
Lookup Cost in a Leveled LSM-Tree

1 run per level

Lookup cost in level $i$, $\mathcal{T}(i)$
- empty ($\alpha_i$)
  \[ \alpha_i \cdot (T_H + T_P + f_p \cdot T_D) \]
- non-empty ($1 - \alpha_i$)
  \[ (1 - \alpha_i) \cdot (T_H + T_P + T_D) \]

\[
\text{cost} \approx \left( L - \frac{1-\alpha}{T-1} \right) \cdot (T_H + T_P) + \left( L - \frac{1-\alpha}{T-1} - 1 + \alpha \right) \cdot (f_p \cdot T_D) + (1 - \alpha) \cdot T_D
\]
Lookup Cost in a Leveled LSM-Tree

1 run per level

Lookup cost in level $i$, $\mathcal{T}(i)$

\[
\begin{align*}
\text{empty} & \quad \alpha_i \cdot (T_H + T_P + f_p \cdot T_D) \\
\text{non-empty} & \quad (1 - \alpha_i) \cdot (T_H + T_P + T_D)
\end{align*}
\]

\[
\text{cost} \approx \left( L - \frac{1-\alpha}{T-1} \right) \cdot (T_H + T_P) + \left( L - \frac{1-\alpha}{T-1} - 1 + \alpha \right) \cdot (f_p \cdot T_D) + (1 - \alpha) \cdot T_D
\]

Bloom filter cost
Lookup Cost in a Leveled LSM-Tree

1 run per level

Lookup cost in level $i$, $\mathcal{T}(i)$
- empty ($\alpha_i$)
  $\alpha_i \cdot (T_{BF} + f_p \cdot T_D)$
- non-empty ($1 - \alpha_i$)
  $(1 - \alpha_i) \cdot (T_{BF} + T_D)$

\[
\text{cost} \approx \left(L - \frac{1-\alpha}{T-1}\right) \cdot (T_H + T_P) + \left(L - \frac{1-\alpha}{T-1} - 1 + \alpha\right) \cdot (f_p \cdot T_D) + (1 - \alpha) \cdot T_D
\]

Bloom filter cost

Data access due to false positives
Lookup Cost in a Leveled LSM-Tree

1 run per level

Lookup cost in level $i$, $\mathcal{T}(i)$
- empty ($\alpha_i$)
  $$\alpha_i \cdot (T_{BF} + f_p \cdot T_D)$$
- non-empty ($1 - \alpha_i$)
  $$(1 - \alpha_i) \cdot (T_{BF} + T_D)$$

$$cost \approx \left( L - \frac{1 - \alpha}{T - 1} \right) \cdot (T_H + T_P) + \left( L - \frac{1 - \alpha}{T - 1} - 1 + \alpha \right) \cdot (f_p \cdot T_D) + (1 - \alpha) \cdot T_D$$

Bloom filter cost

Data access due to false positives

Data access
Lookup Cost in a Leveled LSM-Tree

1 run per level

Lookup cost in level $i$, $\mathcal{T}(i)$
- empty ($\alpha_i$)
  \[ \alpha_i \cdot (T_{BF} + f_p \cdot T_D) \]
- non-empty ($1 - \alpha_i$)
  \[ (1 - \alpha_i) \cdot (T_{BF} + T_D) \]

\[ \propto L \]

\[ \text{cost} \approx \left( L - \frac{1-\alpha}{T-1} \right) \cdot (T_H + T_P) + \left( L - \frac{1-\alpha}{T-1} - 1 + \alpha \right) \cdot (f_p \cdot T_D) + (1 - \alpha) \cdot T_D \]

Hashing accumulates as $L$ grows (larger data size)
Lookup Cost in a Leveled LSM-Tree

1 run per level

Lookup cost in level \( i \), \( T(i) \)
- empty \( \alpha_i \)
  \[ \alpha_i \cdot (T_{BF} + f_p \cdot T_D) \]
- non-empty \( 1 - \alpha_i \)
  \[ (1 - \alpha_i) \cdot (T_{BF} + T_D) \]

\[
\text{cost} \approx \left( L - \frac{1-\alpha}{T-1} \right) \cdot (T_{BF}) + \left( L - \frac{1-\alpha}{T-1} - 1 + \alpha \right) \cdot (f_p \cdot T_D) + (1 - \alpha) \cdot T_D
\]

Hashing is more prominent for empty queries
## Storage Access vs. Hashing

<table>
<thead>
<tr>
<th>Operation</th>
<th>Latency</th>
<th>Normalized</th>
</tr>
</thead>
<tbody>
<tr>
<td>4KB access on SDD</td>
<td>113 µs</td>
<td>706×</td>
</tr>
<tr>
<td>4KB access on PCIe SDD</td>
<td>10 µs</td>
<td>62.5×</td>
</tr>
<tr>
<td><strong>4KB access on emulated NVM</strong></td>
<td>250 ns</td>
<td>1.56×</td>
</tr>
<tr>
<td>4KB access on Memory</td>
<td>160 ns</td>
<td>1×</td>
</tr>
<tr>
<td>Murmur Hash of 1KB</td>
<td>235 ns</td>
<td>1.47×</td>
</tr>
</tbody>
</table>
## Storage Access vs. Hashing

<table>
<thead>
<tr>
<th>Operation</th>
<th>Latency</th>
<th>Normalized</th>
</tr>
</thead>
<tbody>
<tr>
<td>4KB access on SDD</td>
<td>113 $\mu$s</td>
<td>706×</td>
</tr>
<tr>
<td>4KB access on PCIe SDD</td>
<td>10 $\mu$s</td>
<td>62.5×</td>
</tr>
<tr>
<td>4KB access on emulated NVM</td>
<td>250 ns</td>
<td>1.56×</td>
</tr>
<tr>
<td>4KB access on Memory</td>
<td>160 ns</td>
<td>1×</td>
</tr>
<tr>
<td>Murmur Hash of 1KB</td>
<td>235 ns</td>
<td>1.47×</td>
</tr>
</tbody>
</table>

10% faster than NVM
## Storage Access vs. Hashing

<table>
<thead>
<tr>
<th>Operation</th>
<th>Latency</th>
<th>Normalized</th>
</tr>
</thead>
<tbody>
<tr>
<td>4KB access on SDD</td>
<td>113 $\mu$s</td>
<td>706×</td>
</tr>
<tr>
<td>4KB access on PCIe SDD</td>
<td>10 $\mu$s</td>
<td>62.5×</td>
</tr>
<tr>
<td>4KB access on emulated NVM</td>
<td>250 ns</td>
<td>1.56×</td>
</tr>
<tr>
<td>4KB access on Memory</td>
<td>160 ns</td>
<td>1×</td>
</tr>
<tr>
<td>Murmur Hash of 1KB</td>
<td>235 ns</td>
<td>1.47×</td>
</tr>
</tbody>
</table>

10% faster than NVM

What is the time spent hashing as we move to faster devices?
Hashing Overhead in a Leveled LSM-Tree

In SSDs, hashing is over 10% for all empty lookups
Hashing Overhead in a Leveled LSM-Tree

Execution time breakdown

- SSD (113 μs)
- slow PCIe SSD (50 μs)
- fast PCIe SSD (10 μs)
- emulated NVM (250ns)
- memory (160 ns)
- hashing
- I/O

Empty point lookup ratio (α)

Hashing is getting more dominant
Hashing Overhead in a Leveled LSM-Tree

Execution time breakdown

- SSD (113 µs)
- fast PCIe SSD (10 µs)
- memory (160 ns)
- slow PCIe SSD (50 µs)
- emulated NVM (250 ns)
- hashing
- I/O

Empty point lookup ratio ($\alpha$)

Hashing is getting more dominant
Hashing Overhead in a Leveled LSM-Tree

Execution time breakdown

![Chart showing execution time breakdown with various components: SSD (113 μs), slow PCIe SSD (50 μs), fast PCIe SSD (10 μs), emulated NVM (250ns), memory (160 ns), hashing, and I/O.]

Empty point lookup ratio (α)

Hashing is getting more dominant
Hashing Overhead in a Leveled LSM-Tree

Execution time breakdown

Empty point lookup ratio ($\alpha$)

Hashing is getting more dominant
Lookup Cost in a Tiered LSM-Tree

\( T \) runs per level

Lookup cost in level \( i \), \( T'(i) \)
- empty (\( \alpha_i \))
  \[
  \alpha_i \cdot T \cdot (T_H + T_P + f_p \cdot T_D)
  \]
- non-empty (\( 1 - \alpha_i \))
  \[
  (1 - \alpha_i) \cdot \frac{T + 1}{2} \cdot (T_H + T_P) + (1 - \alpha_i) \cdot T_D
  \]

\[
\text{cost} \approx \left(T \cdot L - \frac{T + 1}{2} (1 - \alpha)\right) \cdot (T_{BF}) \quad \text{Bloom filter cost}
\]
\[
+ \left(T \cdot L - (1 - \alpha) \cdot (T + 1)\right) \cdot (f_p \cdot T_D) \quad \text{Data access due to false positives}
\]
\[
+ (1 - \alpha) \cdot T_D \quad \text{Data access}
\]
Lookup Cost in a Tiered LSM-Tree

Lookup cost in level $i$, $\mathcal{T}(i)$

- empty ($\alpha_i$)
  \[
  \alpha_i \cdot T \cdot (T_H + T_P + f_p \cdot T_D)
  \]
- non-empty ($1 - \alpha_i$)
  \[
  (1 - \alpha_i) \cdot \frac{T + 1}{2} \cdot (T_H + T_P) + (1 - \alpha_i) \cdot T_D
  \]

Cost:

\[
\text{cost} \approx \left( T \cdot L - \frac{T + 1}{2} (1 - \alpha) \right) \cdot (T_H + T_P) + \left( T \cdot L - (1 - \alpha) \cdot (T + 1) \right) \cdot (f_p \cdot T_D) + (1 - \alpha) \cdot T_D
\]

$\propto T \cdot L$

Bloom filter cost

Data access due to false positives

Data access
Hashing Overhead in a Tiered LSM-Tree

Similar, but hashing is more pronounced.
How can we reduce the hashing overhead in LSM-trees?
Hash Sharing in Leveled LSM-Trees

(a) Query path

(b) Hashing in a query

(c) Shared hashing in a query
Hash Sharing in Leveled LSM-Trees

The same hash function is calculated multiple times which brings CPU overhead.
Hash Sharing in Leveled LSM-Trees

The same hash function is calculated multiple times which brings CPU overhead.

A single hash digest can be reused to avoid expensive hash calculations.
Theoretical Gain w.r.t. Evolving Storage Devices

Gain = \frac{\text{cost} - \text{cost}_{\text{share}}}{\text{cost}} \times 100\%

Gain

0% 10% 20% 30% 40% 50%

≈0% 1% 6% 40%

HDD SSD PCIe SSD NVM

Storage devices
Theoretical Gain w.r.t. Evolving Storage Devices

\[
Gain = \left( \frac{\text{cost} - \text{cost}_{\text{share}}}{\text{cost}} \right) \times 100\%
\]

The gain increases rapidly for faster storage devices.
Build an LSM prototype (RocksDB’s fast local BF).

1M point queries (report avg latency of 5 experiments)

Use PCIe SSD (10 μs) with direct I/O by default
Impact of the key size

Uniform Query Distribution

Workload: Entry size: 2KB, #Entries: 11M
Tuning: Bits per key: 10, Size ratio: 10, Storage: PCIe SSD
Impact of the key size

*Workload:* Entry size: 2KB, Entries: 11M
*Tuning:* Bits per key: 10, Size ratio: 10, Storage: PCIe SSD
Impact of the key size

**Workload:** Entry size: 2KB, #Entries: 11M

**Tuning:** Bits per key: 10, Size ratio: 10, Storage: PCIe SSD
Impact of the key size

**Workload:** Entry size: 2KB, #Entries: 11M  
**Tuning:** Bits per key: 10, Size ratio: 10, Storage: PCIe SSD

As key size grows, the gain increases up to 20%
Impact of the key size

Workload: Entry size: 2KB, #Entries: 11M
Tuning: Bits per key: 10, Size ratio: 10, Storage: PCIe SSD

As key size grows, the gain increases up to 20%

For skewed empty query workload, the gain increases up to 60%
Impact of #levels

Workload:
Key size: 64B
Entry size: 1KB
#Entries: 22M

Tuning:
Bits per key: 10
Size ratio: 2
Storage: PCIe SSD
Impact of #levels

The gain initially increases as #level grows, and then plateaus.

Workload:
- Key size: 64B
- Entry size: 1KB
- #Entries: 22M

Tuning:
- Bits per key: 10
- Size ratio: 2

Storage:
- PCIe SSD
**Impact of storage device**

- **Workload:**
  - Key size: 64B
  - Entry size: 1KB
  - #Entries: 22M

- **Tuning:**
  - Bits per key: 10
  - Size ratio: 10
Hash sharing leads to higher gain for faster storage.

Workload:
- Key size: 64B
- Entry size: 1KB
- #Entries: 22M

Tuning:
- Bits per key: 10
- Size ratio: 10
Impact of the I/O cost of empty queries

**Workload:**
Entry size: 1KB
#Entries: 22M
Empty query ratio (\(\alpha\)) : 1

**Tuning:**
Bits per key: 10
Size ratio: 10

For slow storage, high I/O cost of empty queries leads to smaller gain.
Impact of the I/O cost of empty queries

**Workload:**
- Entry size: 1KB
- #Entries: 22M
- Empty query ratio ($\alpha$): 1

**Tuning:**
- Bits per key: 10
- Size ratio: 10

For slow storage, high I/O cost of empty queries leads to smaller gain.

For fast storage, the gain does not depend on the cost of empty queries.

---

**Diagram:**
- PCIe SSD(D) vs RAM
- Gain vs Bits per key (BPK)
- High I/O cost of empty queries
- Low I/O cost of empty queries
Impact of the I/O cost of empty queries

**Workload:**
- Entry size: 1KB
- #Entries: 22M
- Empty query ratio ($\alpha$): 1

**Tuning:**
- Bits per key: 10
- Size ratio: 10

**Fast storage** leads to high gain. Even for slower storage, if the cost of empty queries is low (low FPR), the gain is high.
Impact of empty lookup ratio ($\alpha$)

Storage: PCIe SSD (D)

Workload: Entry size: 1KB, #Entries: 22M
Tuning: Bits per key: 10, Size ratio: 10
Impact of empty lookup ratio ($\alpha$)

Storage: PCIe SSD (D)

Workload: Entry size: 1KB, #Entries: 22M
Tuning: Bits per key: 10, Size ratio: 10

The gain on PCIe SSD increases as $\alpha$ increases
Impact of empty lookup ratio ($\alpha$)

**Storage: PCIe SSD (D)**

- state-of-art
- Hash Sharing
- BF(hash+probe)
- data
- other

**Storage: RAM disk**

- state-of-art
- Hash Sharing
- BF(hash+probe)
- data
- other

The gain on PCIe SSD increases as $\alpha$ increases

The benefit is pronounced when it comes to a RAM disk

**Workload:** Entry size: 1KB, #Entries: 22M

**Tuning:** Bits per key: 10, Size ratio: 10
The gain on PCIe SSD increases as $\alpha$ increases.

Overall, hash sharing has more impact for faster devices which is further exacerbated for empty queries.

The benefit is pronounced when it comes to a RAM disk.
Conclusion

- BFs dominate LSM query latency for fast storage
Conclusion

- BF\textsuperscript{s} dominate LSM query latency for fast storage

- Develop a query cost model to quantify and predict the amount of time on hashing and data accessing
Conclusion

- BFs dominate LSM query latency for fast storage

- Develop a query cost model to quantify and predict the amount of time on hashing and data accessing

- Reduce hashing, by sharing it across BFs and levels, leading to performance gains up to 40%
Conclusion

- BFs dominate LSM query latency for fast storage
- Develop a query cost model to quantify and predict the amount of time on hashing and data accessing
- Reduce hashing, by sharing it across BFs and levels, leading to performance gains up to 40%

Thank you!

https://github.com/BU-DiSC/BF-Shared-Hashing