Don’t Thrash: How to Cache Your Hash in Flash

Michael A. Bender
Bradley C. Kuszmaul
Pradeep Shetty

Martin Farach-Colton
Dzejla Medjedovic
Richard P. Spillane

Rob Johnson
Pablo Montes
Erez Zadok
A Bloom Filter (BF) is a bit-array + $k$ hash functions.

- Each element is hashed to $k$ out of $m$ positions in a bit-array. Here $k=2$.
- The BF stays in RAM because it only stores a few bits per element independent of their sizes.

Elements stored in the Bloom filter

Bit-array

0 1 0 1 1
Bloom Filter (approx membership queries)

- A BF is a bit-array + $k$ hash functions
- The BF stays in RAM because it only stores a few bits per element independent of their sizes.
Don't Thrash: How to Cache Your Hash in Flash

Bloom Filter lookups

False positive

Cache

0 1 0 1 1

Store

A B

Bounded rate of false positives: \( p_{FP}(x) \approx (1 - e^{kn/m})^k \)

No false negatives.
Don't Thrash: How to Cache Your Hash in Flash

Bounded rate of false positives:

No false negatives.

Fast way to say “No” without going to disk.

The Bloom filter is a well-loved hammer in a system-builders toolbox.
Flash (e.g., SSD) is bigger & cheaper than RAM, faster than disk.

- Good place to store Bloom filter, in between disk and RAM.

Bloom filters even for moderate-sized data sets may be too large for RAM.

- Example: 8TB of 512B keys needs 16GB of RAM for a ~1% BF.
We should think about Flash the way we think about disk, except that random I/O is faster.

- Two-levels of memory.
- Two parameters: block-size $B$, memory-size $M$.

**Metric: # of block transfers.**

- Block transfers are faster in flash than disk, but the issues are similar.
We should think about Flash the way we think about disk, except that random I/O is faster.

- Two-levels of memory.
- Two parameters: block-size $B$, memory-size $M$.

**Metric: # of block transfers.**

- Block transfers are faster in flash than disk, but the issues are similar.

(More accurate SSD models exist [Ajwani, Beckmann, Jacob, Meyer, and Moruz 2009] but aren’t necessary here.)
Thrashing in Bloom filter implemented in Flash.

- Setting $k$ random bits to 1 causes $O(k)$ random writes.
- OK in RAM, expensive in Flash.

We cannot have an efficient data structure without working around this issue.
Cascade Filter (CF), a BF replacement optimized for fast insertions/deletions on flash.

Write-optimized performance:

- 670,000 inserts/sec (3000x of Bloom, 40x best alternative)
- 530 lookups/sec (1/3x of best alternative)

The Cascade Filter is based upon Quotient Filters (QFs) instead of BF

- QFs have better access locality.
- QFs support deletes.

We can efficiently merge two QFs into a larger QF, which is important for fast insertions.
We just presented this result at HotStorage.

- Not an algorithms conference.
- The storage community is grappling with difficult I/O issues. Feeling tooth pain.
- Community understands that I/O-efficient algorithms can help.
We just presented this result at HotStorage.

- Not an algorithms conference.
- The storage community is grappling with difficult I/O issues. Feeling tooth pain.
- Community understands that I/O-efficient algorithms can help.

Of interest to several write-optimized storage systems

- Apache: HBase
- Facebook: Cassandra
- Google: BigTable
- Tokutek: TokuDB
Don't Thrash: How to Cache Your Hash in Flash

Thrashing in Bloom filter implemented in Flash.

- Setting $k$ random bits to 1 causes $O(k)$ random writes.
- OK in RAM, expensive in Flash.

We cannot have an efficient data structure without working around this issue.
Don't Thrash: How to Cache Your Hash in Flash

Use a two-step hash to shave off $k$

- Store a separate Bloom filter in each flash page.
- First hash to a particular page/Bloom filter.
- Perform the $k$ writes.

This helps a little.

- 1 I/O per insertion. We want <1 I/O per insert.
Amortize the I/O cost over many bit writes.

- This helps a lot  [Canim, Mihaila, Bhattacharjee, Lang, Ross, 2010]
Buffering works well when Flash isn’t too large compared to RAM.

Buffering degrades as flash size grows, approaching ~1 I/O per insert.

We’re interested in large datasets and fast insertions (i.e., when buffering doesn’t work)
Cascade Filter (CF), a BF replacement optimized for fast insertions/deletions on flash.

Write-optimized performance:
- 670,000 inserts/sec (40x of other variants)
- 530 lookups/sec (1/3x of other variants)

The Cascade Filter is based upon Quotient Filters (QFs) instead of BF
- QFs have better access locality.
- QFs support deletes.

We can efficiently merge two QFs into a larger QF, which is important for fast insertions.
Idea of Quotienting

- Take a fingerprint \( h(u) \) for each element \( u \).
- Store fingerprints *compactly* in a hashtable.
  (Next slide: how to store compactly.)
Idea of Quotienting

- Take a fingerprint $h(u)$ for each element $u$.
- Store fingerprints compactly in a hashtable. (Next slide: how to store compactly.)

Only source of False Positives

- Two distinct elements $u$ and $v$, where $h(u)=h(v)$.
- If $u$ is stored and $v$ isn’t, Query($v$) gives a false positive.
Quotienting  [Knuth]
(Alternative to Bloom Filters)

$u \rightarrow h(u) = a(u) \bmod m + b(u)$

- $a(u) = \text{location in hash table}$
- $b(u) = \text{data stored in hash table}$
Quotienting [Knuth]
(Alternative to Bloom Filters)

Collisions in the hash table?

$a(u) = \text{location in hash table}$

$b(u) = \text{data stored in hash table}$
Quotienting  [Knuth]
(Alternative to Bloom Filters)

Collisions in the hash table?
- Linear probing?
- Yes, but be careful.

\[ h(u) = a(u) \log m + b(u) \]

\[ a(u) = \text{location in hash table} \]
\[ b(u) = \text{data stored in hash table} \]
Ex: 6 bit hash. 3 bits for address, 3 for data.

<table>
<thead>
<tr>
<th>Address</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>001</td>
</tr>
<tr>
<td>001</td>
<td>011</td>
</tr>
<tr>
<td>010</td>
<td></td>
</tr>
<tr>
<td>011</td>
<td>010</td>
</tr>
<tr>
<td>100</td>
<td>111</td>
</tr>
<tr>
<td>101</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td>100</td>
</tr>
<tr>
<td>111</td>
<td></td>
</tr>
</tbody>
</table>
Ex: 6 bit hash. 3 bits for address, 3 for data.

```
000 001
001 011
010 010
011 111
100 111
101 100
110 100
111
```

Does this element represent 001011 or 000011?
Ex: 6 bit hash. 3 bits for address, 3 for data.

- Does this element represent 001011 or 000011?
- Does this element represent 100111 or 011111?
Need control bits to

- store distance $\Delta$ between “target” and stored location in the hash table;
- Indicate which array positions are filled.

Solutions:

- Naive: $O(\log \log n)$, since $\Delta = O(\log n)$ w.h.p.
- Can do with 2 control bits, but our implementation was too slow.
- We chose 3 bits.
- How much better can we do than 2?
QF: alternative to Bloom filter.

- Supports deletes as well as inserts.
- 1 I/O per insert/delete/lookup in expectation
- Just a compact, and easily implementable data structure for storing hashes.

Next slide: merge exponentially increasing Quotient Filters to generate the Cascade Filter.
Goal: $<<1$ I/O per insert/delete.

Use a standard approach for write-optimized structures:

- cascading QFs of exponentially increasing size
- merge QFs by sequential scans

[O'Neil, Cheng, Gawlick, O'Neil 96], [Buchsbaum, Goldwasser, Venkatasubramanian, Westbrook 00], [Argel 03], [Graefe 03], [Brodal, Fagerberg 03], [Bender, Farach, Fineman, Fogel, Kuszmaul, Nelson'07], [Brodal, Demaine, Fineman, Iacono, Langerman, Munro 10]
Goal: <<1 I/O per insert/delete.

Use one standard approach for write-optimized structures:

- cascading QFs of exponentially increasing size
- merge QFs by sequential scans

[O'Neil, Cheng, Gawlick, O'Neil 96], [Buchsbaum, Goldwasser, Venkatasubramanian, Westbrook 00], [Argel 03], [Graefe 03], [Brodal, Fagerberg 03], [Bender, Farach, Fineman, Fogel, Kuszmaul, Nelson’07], [Brodal, Demaine, Fineman, Iacono, Langerman, Munro 10]
Goal: \(<<1\) I/O per insert/delete.

Use one standard approach for write-optimized structures:

- cascading QFs of exponentially increasing size
- merge QFs by sequential scans
Goal: <<1 I/O per insert/delete.

Use one standard approach for write-optimized structures:

- cascading QFs of exponentially increasing size
- merge QFs by sequential scans
Goal: <<1 I/O per insert/delete.

Use one standard approach for write-optimized structures:

- cascading QFs of exponentially increasing size
- merge QFs by sequential scans
Goal: $<1$ I/O per insert/delete.

Use one standard approach for write-optimized structures:

- cascading QFs of exponentially increasing size
- merge QFs by sequential scans

[O'Neil, Cheng, Gawlick, O'Neil 96], [Buchsbaum, Goldwasser, Venkatasubramanian, Westbrook 00], [Argel 03], [Graefe 03], [Brodal, Fagerberg 03], [Bender, Farach, Fineman, Fogel, Kuszmaul, Nelson'07], [Brodal, Demaine, Fineman, Iacono, Langerman, Munro 10]
Goal: \(<\!<1\) I/O per insert/delete.

Use one standard approach for write-optimized structures:

- cascading QFs of exponentially increasing size
- merge QFs by sequential scans

[O'Neil, Cheng, Gawlick, O'Neil 96], [Buchsbaum, Goldwasser, Venkatasubramanian, Westbrook 00], [Argel 03], [Graefe 03], [Brodal, Fagerberg 03], [Bender, Farach, Fineman, Fogel, Kuszmaul, Nelson'07], [Brodal, Demaine, Fineman, Iacono, Langerman, Munro 10]
Goal: <<1 I/O per insert/delete.

Use one standard approach for write-optimized structures:

- cascading QFs of exponentially increasing size
- merge QFs by sequential scans
Goal: <<1 I/O per insert/delete.

Use one standard approach for write-optimized structures:

- cascading QFs of exponentially increasing size
- merge QFs by sequential scans

\[ \text{Insert/delete: } O((\log \frac{N}{M})/B) \]
\[ \text{Query: } O(\log \frac{N}{M}) \]
Insertion Throughput

Number of Fingerprints Inserted

Seconds

Up is better

Don't Thrash: How to Cache Your Hash in Flash
Insertion Throughput

Peak append throughput: 8.4MB/S

Large Merges

Thruput much higher: 40x higher than BBF
3000x higher than BF

Up is better
Lookup Throughput

- CF: 530 lkus/sec
- Traditional BF: 1600 lkus/sec
- Elevator BF: 1600 lkus/sec

Don't Thrash: How to Cache Your Hash in Flash
Cascade Filter (CF), a BF replacement optimized for fast insertions/deletions on flash.

Write-optimized performance:
- 670,000 inserts/sec (3000x of Bloom, 40x best alternative)
- 530 lookups/sec (1/3x of best alternative)

The Cascade Filter is based upon Quotient Filters (QFs) instead of BF
- QFs have better access locality.
- QFs support deletes.

We can efficiently merge two QFs into a larger QF, which enables fast insertions.
Write optimized data structures are solving problems where people fill pain

- indexing in databases
- metadata maintenance on cloud/parallel file systems
- creating of tens of thousands of microfiles/sec/disk
- deduplication

Cascade filters (Approx membership queries) helps many of these applications

- queries on write-optimized systems
- insertions when there are uniqueness constraints