Morton filters: fast, compressed sparse cuckoo filters

Alex D Breslow and Nuwan S Jayasena

Presented by William Qian

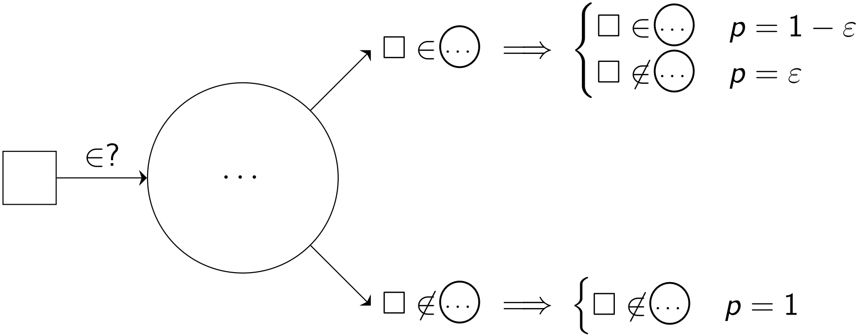
2020 April 30

6.886 Spring 2020

Overview

1. Background
2. Morton filters
3. Evaluation
4. Discussion

Approximate set membership data structures

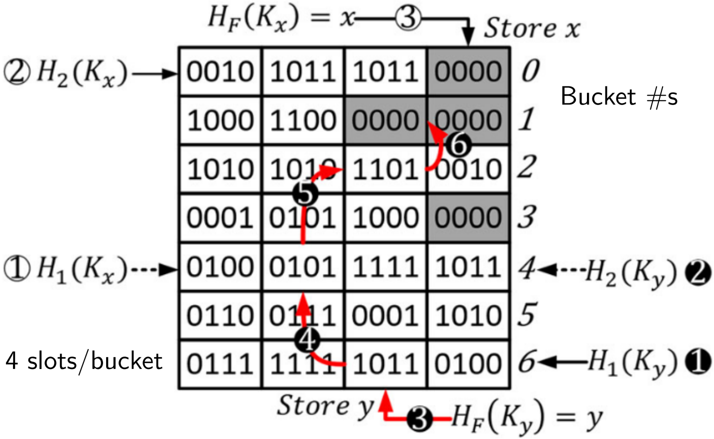


Examples: Bloom filters, Cuckoo filters [3], Morton filters [1, 2]...

Cuckoo filters

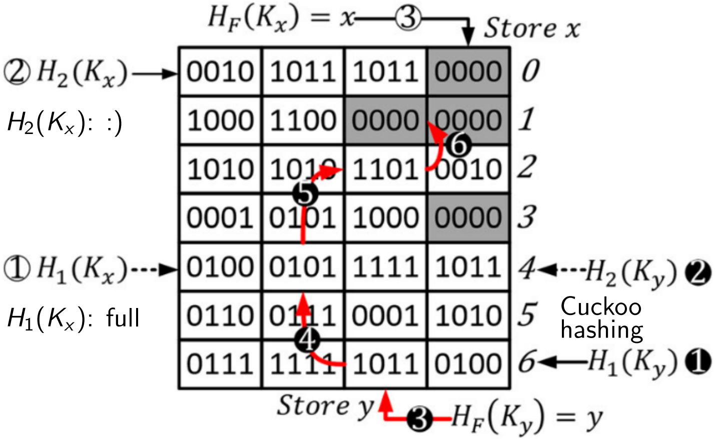
Fingerprints are fixed-width hashes of keys using H_F

Buckets are determined by either H_1 or H_2



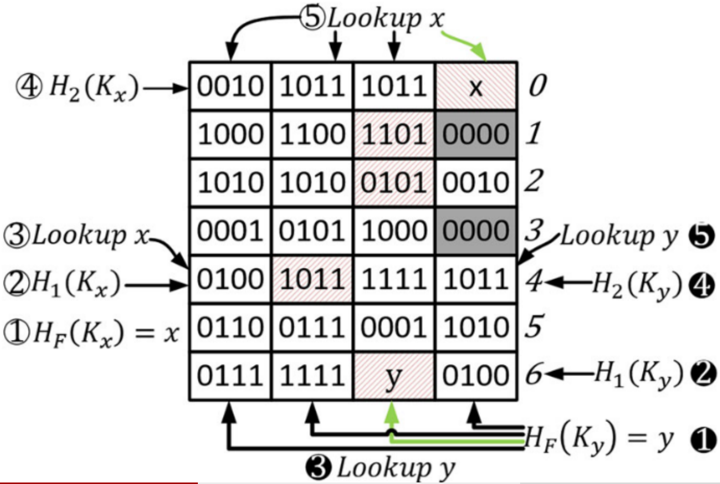
Cuckoo filters: insertions

Pick empty slot in either bucket
No available slots: evict an entry and cascade via Cuckoo hashing



Cuckoo filters: lookups

Look in both buckets for matching fingerprint
Found match: likely in set; no match: not in set



Morton filters: overview

Morton filters (MFs) [1, 2] are like Cuckoo filters (CFs), but MFs:

- Bias toward one hash function over the other
- Use a compressed block store
- Require $2x$ buckets, instead of 2^x buckets

Morton filters: primacy

Preferentially hash using H_1; H_2 is the backup

- Lookups generally require only one hash (and thus, cache line)

Morton filters: compressed block store

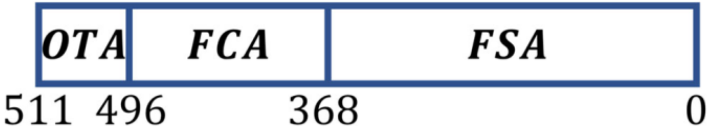


Fig. 3 A sample block in an MF that is performance-optimized for 512-bit cache lines. The block has a 46-slot FSA with 8-bit fingerprints, a 64-slot FCA with 2-bit fullness counters (64 3-slot buckets), and a 16-bit OTA with a bit per slot

- Sparseness $\implies$ not all slots will be used
- Bitmaps to maintain meta information
- FSA: fingerprint storage array. Contains fixed-width fingerprints.
- FCA: fullness counter array. b bits/counter, $2^b - 1$ slots/bucket.
- OTA: overflow tracking array. 1 indicates block/bucket overflow.

Morton filters: compressed block store

Block overflows occur when the FSA has run out of space

- Evicts some (any) fingerprint

Bucket overflows occur when the bucket's FCA has reached its max

- Evicts a fingerprint in the bucket

When a bucket's OTA bit is set, it indicates that if a key hashed there with H_1 isn't found in the bucket, we should look at its H_2 bucket as well.

Morton filters: compressed block store

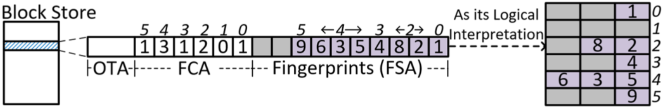


Fig. 4 An MF's Block Store and a sample block's compressed format and logical interpretation, with corresponding buckets labeled 0 to 5. The FCA and FSA state dictates the logical interpretation of the block. Buckets and fingerprints are ordered right to left to be consistent with logical shift operations

Morton filters: parity-based partial key hashing

$$
\begin{aligned}
H_1(K) &= bucket(\mathcal{H}(K), n) \\
H_2(K) &= bucket(H_1(K) + (-1)^{H_1(K)\&1} \cdot offset(H_{fp}(K)), n) \\
H'(\beta, H_{fp}(K)) &= bucket(\beta + (-1)^{\beta\&1} \cdot offset(H_{fp}(K)), n) \\
offset(fp) &= (B + (fp \bmod OFFSET\_RANGE))|1 \\
bucket(x, n) &= (x + n) \bmod n
\end{aligned}
$$

(*bucket* is implemented to avoid division instructions like / and %)

Morton filters: parity-based partial key hashing

$$\begin{aligned} H'(H_1(K), H_{fp}(K)) &= bucket(H_1(K) + (-1)^{H_1(K) \& 1} \cdot offset(H_{fp}(K)), n) \\ &= H_2(K) \end{aligned}$$

Morton filters: parity-based partial key hashing

offset() is always odd, and n is always even:

$$\begin{aligned} H_2(K)\&1 &= bucket(H_1(K) + (-1)^{H_1(K)\&1} \cdot offset(H_{fp}(K)), n)\&1 \\ &= (H_1(K) + (-1)^{H_1(K)\&1} \cdot offset(H_{fp}(K)))\&1 \\ &= (H_1(K)\&1) \wedge ((-1)^{H_1(K)\&1} \cdot offset(H_{fp}(K))\&1) \\ &= (H_1(K)\&1) \wedge ((-1)^{H_1(K)\&1}\&1) \\ &= (H_1(K)\&1) \wedge 1 \\ &= \sim (H_1(K)\&1) \end{aligned}$$

Morton filters: parity-based partial key hashing

$$
\begin{aligned}
H'(H_2(K), H_{fp}(K)) =& bucket(H_2(K) + (-1)^{H_2(K)\&1} \cdot offset(H_{fp}(K)), n) \\
=& bucket(\\
& H_2(K) + (-1)^{(H_1(K)\&1)+1} \cdot offset(H_{fp}(K)), n) \\
=& bucket(H_2(K) - (-1)^{H_1(K)\&1} \cdot offset(H_{fp}(K)), n) \\
=& bucket(H_1(K) + (-1)^{H_1(K)\&1} \cdot offset(H_{fp}(K)) \\
& - (-1)^{H_1(K)\&1} \cdot offset(H_{fp}(K)), n) \\
=& bucket(H_1(K), n) \\
=& H_1(K)
\end{aligned}
$$

Morton filters: parity-based partial key hashing

$$H'(H_2(K), H_{fp}(K)) = H_1(K); H'(H_1(K), H_{fp}(K)) = H_2(K)$$

$\therefore$ applying H' to an already-inserted key swaps its bucket.

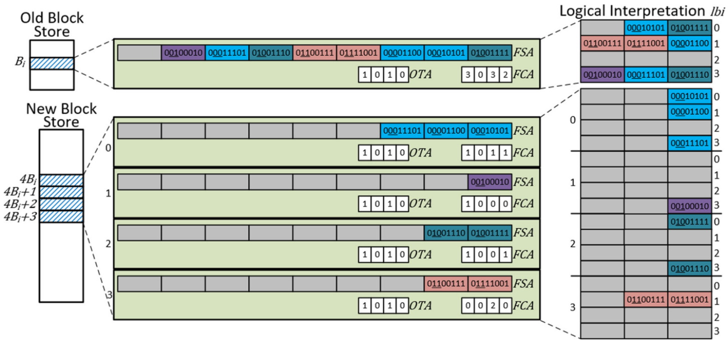
Morton filters: other features

Block full array (BFA) is another bit vector that stores information about which blocks are full

- Insertions can query the BFA to avoid cascading evictions
- Extra overhead for deletes
- Only useful at high loads (FSA generally quite full)

Morton filters: other features

Resizing: MFs can only be resized by powers of 2



- Use significant bits of the fingerprint to assign keys to child buckets

Environment

- AMD Ryzen Threadripper 1950X
  - 2 sockets, 8 cores each, hyperthread enabled
- 512-bit blocks
  - 3-slot: 16-bit OTA, 128-bit (64×2) FCA, 46-slot FSA, 8-bit fp
  - 7-slot: 17-bit OTA, 63-bit (21×3) FCA, 54-slot FSA, 8-bit fp
  - 15-slot: 17-bit OTA, 63-bit (21×3) FCA, 54-slot FSA, 8-bit fp
- Benchmarks: MF (this work), CF (12 bits)

Error rate

Error rate roughly matches projected error rates

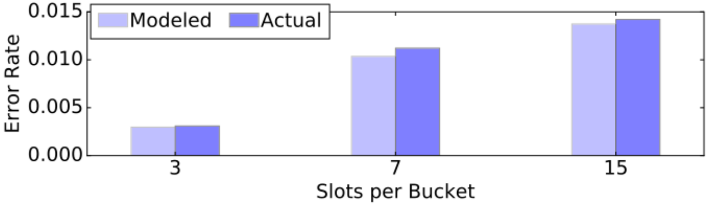


Fig. 11 The MF implementation's false positive rate closely matches Eq. 5. All MFs have a block load factor of 0.95. The MF with 3-slot buckets uses 128 bits for its FCA versus the 7- and 15-slot that use 63 and 64 bits, respectively

Throughput

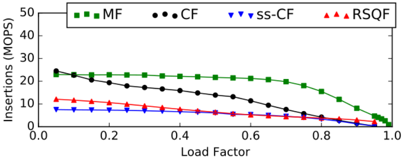


Fig. 14 An MF's insertion throughput is 0.94× to 20.8× that of a CF

(a) Inserts

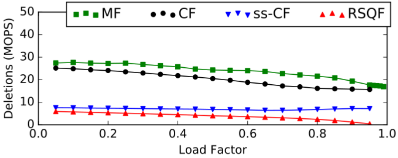


Fig. 16 An MF's deletion throughput is 1.1× to 1.3× higher than that of a CF

(b) Deletes

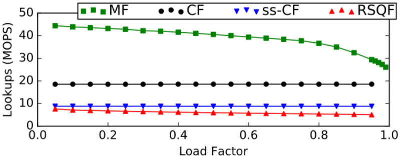


Fig. 12 An MF's positive lookup throughput is about 1.6× to 2.4× higher than a CF's

(c) Positive lookups

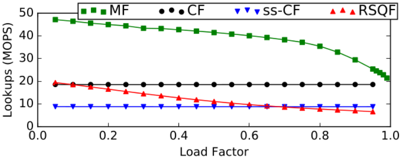
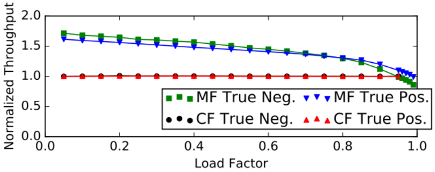


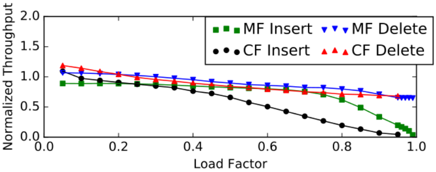
Fig. 13 An MF's negative lookup throughput is about 1.3× to 2.5× higher than a CF's

(d) Negative lookups

Throughput (Intel)



(a) Lookup Throughput



(b) Update Throughput

Fig. 26 On a Skylake-X server, MF lookup throughput is on par with to nearly 1.8× higher than a CF's. MF deletion throughput is about 0.90× to 1.1× a CF's. MF insertion throughput is 0.82× to 4.8× that of a CF. Results are normalized to a CF's lookup throughput on a Skylake-X CPU

Block full array

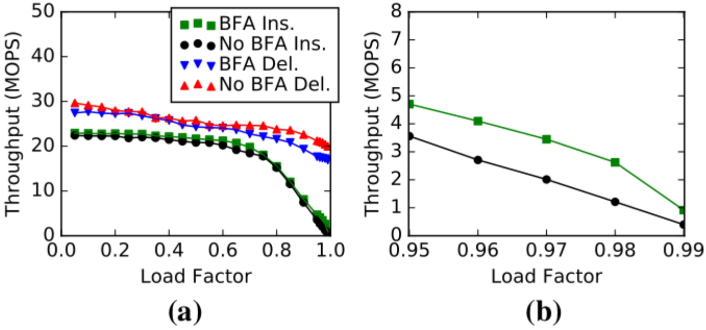


Fig. 21 MF insertion and deletion throughput with and without the BFA enabled. **b** zooms in on the lower right corner of (**a**)

References

Alex D Breslow and Nuwan S Jayasena.
Morton filters: faster, space-efficient cuckoo filters via biasing, compression, and decoupled logical sparsity.
Proceedings of the VLDB Endowment, 11(9):1041–1055, 2018.

Alex D Breslow and Nuwan S Jayasena.
Morton filters: fast, compressed sparse cuckoo filters.
The VLDB Journal, pages 1–24, 2019.

Bin Fan, Dave G Andersen, Michael Kaminsky, and Michael D Mitzenmacher.
Cuckoo filter: Practically better than bloom.
In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*, pages 75–88, 2014.

Takeaways

- Spatial underutilization is expensive!
- This is an interesting metadata design
- Biasing toward one hash function reduces cache costs
- Parity tricks are really cool :)
- Morton filters are competitive with cuckoo filters, and more memory efficient

Discussion

1. Is NUMA important here? How might a NUMA-aware implementation work?
2. What concurrency overheads might exist with this solution?
3. This is published in VLDB(J), which ostensibly means it should be somewhat database-related. What are some implementations/optimizations that might be useful if we wanted to implement this in a distributed memory model?