Don't Thrash: How to Cache Your Hash on Flash

M.A. Bender, M. Farach-Colton, R. Johnson, R. Kraner, B.C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R.P. Spillane, E. Zadok

Summary

We give three Approximate Membership Query (AMQ) data structures:

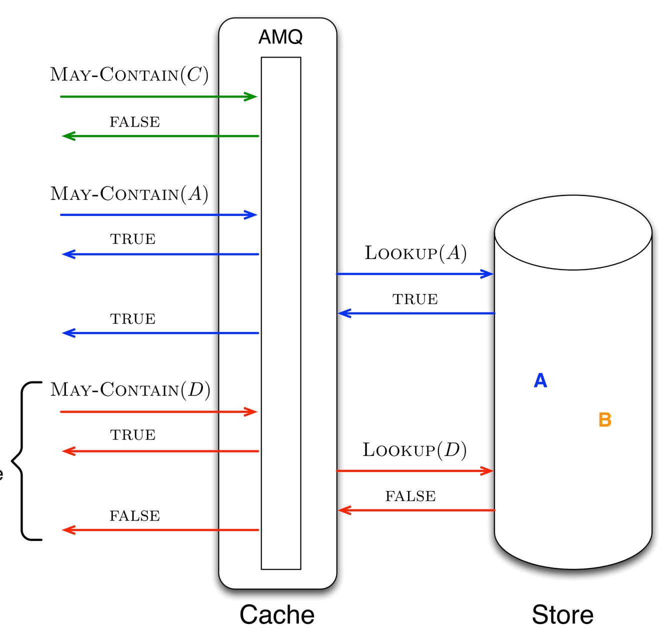
- Quotient Filter (QF).
- Buffered Quotient Filter (BQF).
- Cascade Filter (CF).

QF is an in-RAM alternative to the Bloom Filter (BF). BQF and CF are external-memory AMQ data structures built upon QF.

Background and Related Work

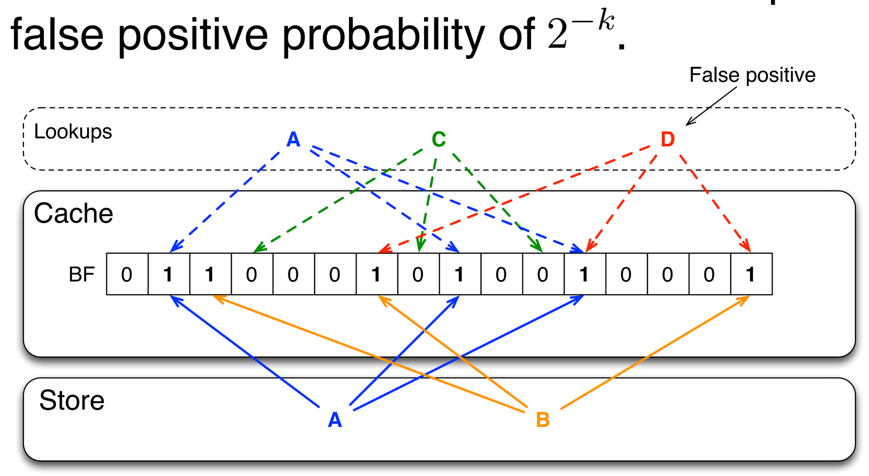
Approximate Membership Query (AMQ) data structure

- Is used to avoid unnecessary and expensive disk lookups for non-existent elements.
- Is a space-efficient representation of a set supporting:
  - $\text{INSERT}(x)$.
  - $\text{MAY-CONTAIN}(x)$.
- Can falsely report that an element is in the set when it is not (false positive).



Bloom Filter (BF) [3]

- Is one of the most widely used AMQ data structures.
- Sets/checks k random bits on an insert/lookup.
- Has a false positive probability of 2^{-k}.



An external-memory AMQ data structure is needed because

- BF size is set upfront and is directly proportional to the maximum number of elements expected.
- If the BF outgrows RAM, its performance decays because of poor data locality.

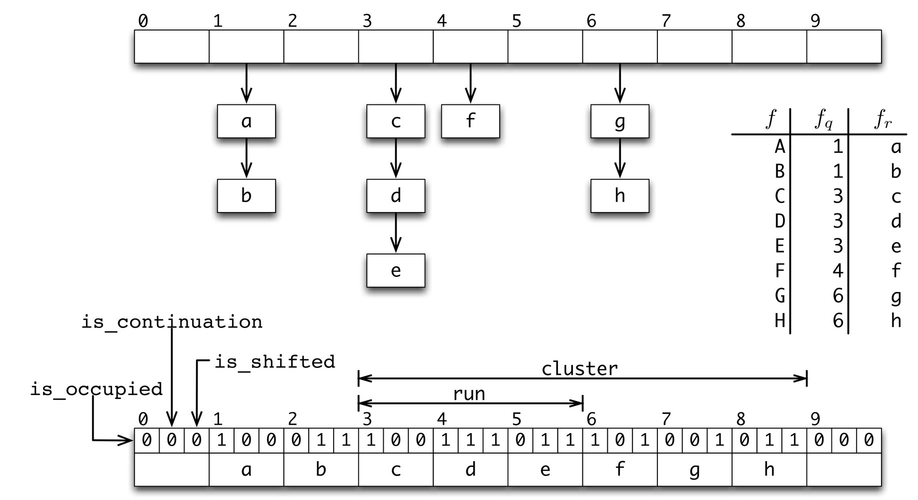
Previous attempts to improve BF scalability

- Storing BF on SSD [4, 6, 8].
- Buffering [4, 6, 8].
- Hash localization [4, 6].
- Multi-layered design [8].
- Elevator Bloom Filter (EBF) *included as a baseline.*
- Buffered Bloom Filter (BBF) [4].
- Forest-structured Bloom Filter (FBF) [8].

Quotient Filter (QF)

QF

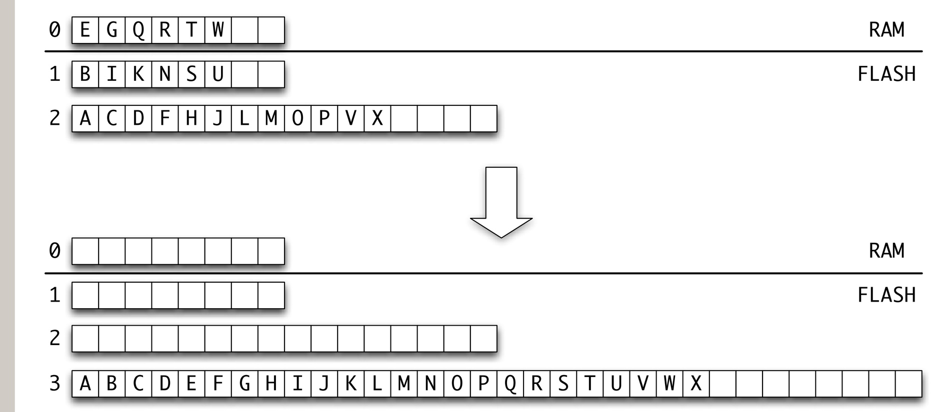
- Is a cache-friendly AMQ data structure.
- Maintains a p-bit fingerprint, f, for each element in an open hash table with $m = 2^q$ buckets using a technique called *quotienting* [7, Sec. 6.4 ex. 13]:
  - The fingerprint is partitioned into its r least significant bits, f_r, and its $q = p - r$ most significant bits, f_q.
  - f_r is stored in bucket f_q.
- Compactly stores the hash table in an array of $(r+3)$-bit items using linear probing as in [5]. We use three meta-data bits per slot to enable decoding.
- Has a false positive probability of 2^{-r}.



Advantages

- Supports correct deletes.
- Supports in-order iteration. This enables:
  - Dynamically resizing without rehashing.
  - Efficient merging of two or more QFs.

Buffered Quotient Filter (BQF) / Cascade Filter (CF)



Asymptotics

In the external memory model [1], the number of I/Os performed by BQF and CF is

	$\text{INSERT}(x)$ amortized	$\text{MAY-CONTAIN}(x)$ expected
BQF	$O(\min\{1, N/MB\})$	$O(1)$
CF	$O(\log_b(N/M)/B)$*	$O(\log_b(N/M))$

*Can be deamortized.

BQF and CF are external-memory AMQ data structures.

BQF

- Maintains one QF in RAM as a buffer and one larger QF on SSD.

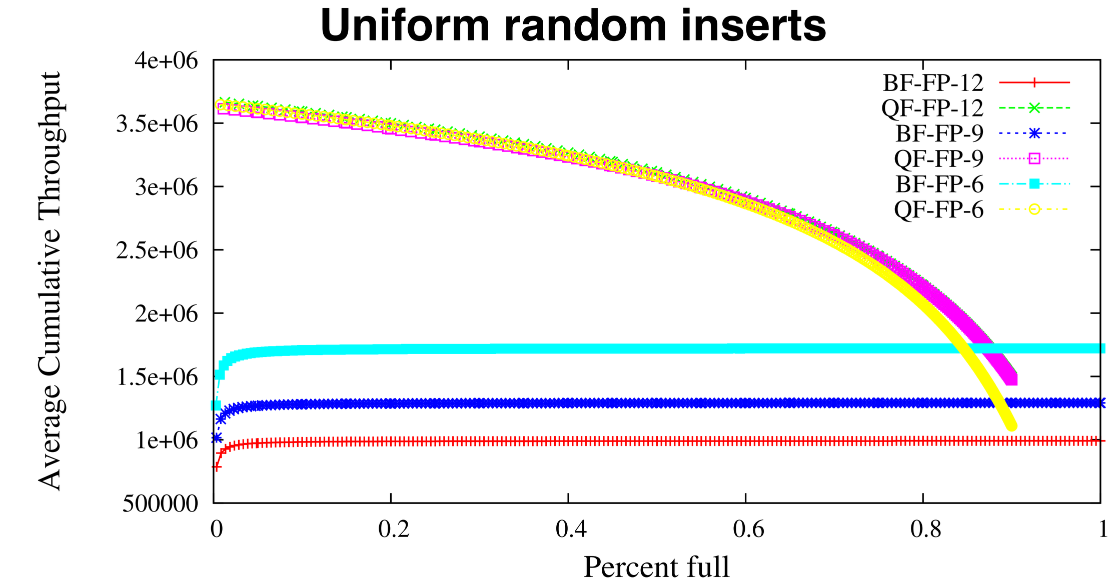
CF

- Is based on COLA [2].
- Maintains an in-memory QF and $\ell = \log_b(N/M)$ QFs of exponentially increasing size on SSD.
- Has a false positive probability of at most twice that of its largest QF.
- Has a customizable branching factor, b, that provides a tradeoff between insert and lookup performance.

Experimental Results

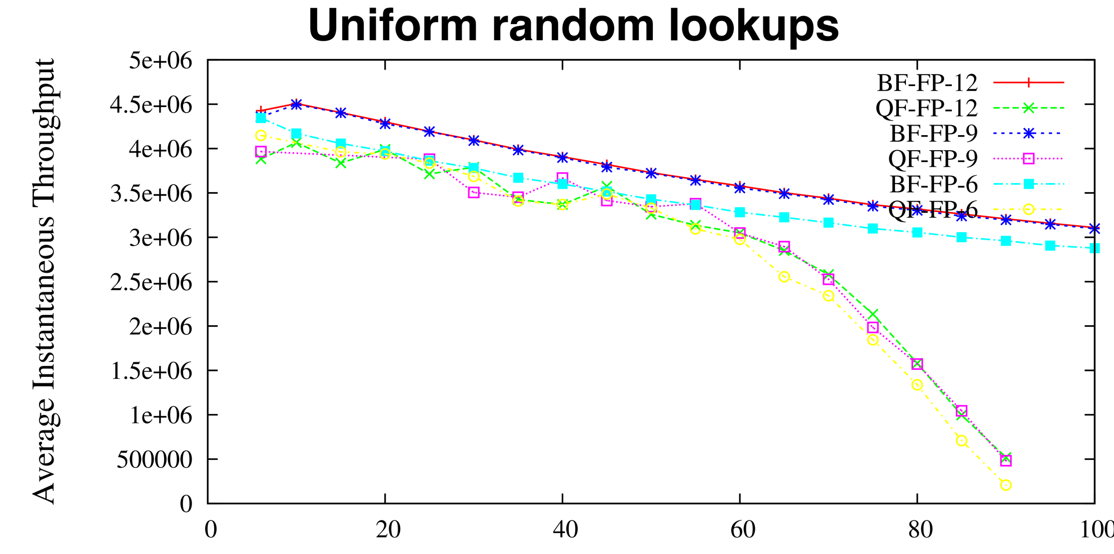
In-RAM

Uniform random inserts



- QF substantially outperforms BF until about 80% full.
- BF throughput is independent of its occupancy, but degrades as the false positive rate goes down.
- QF throughput is unaffected by the false positive rate, but gets slower as it becomes full.

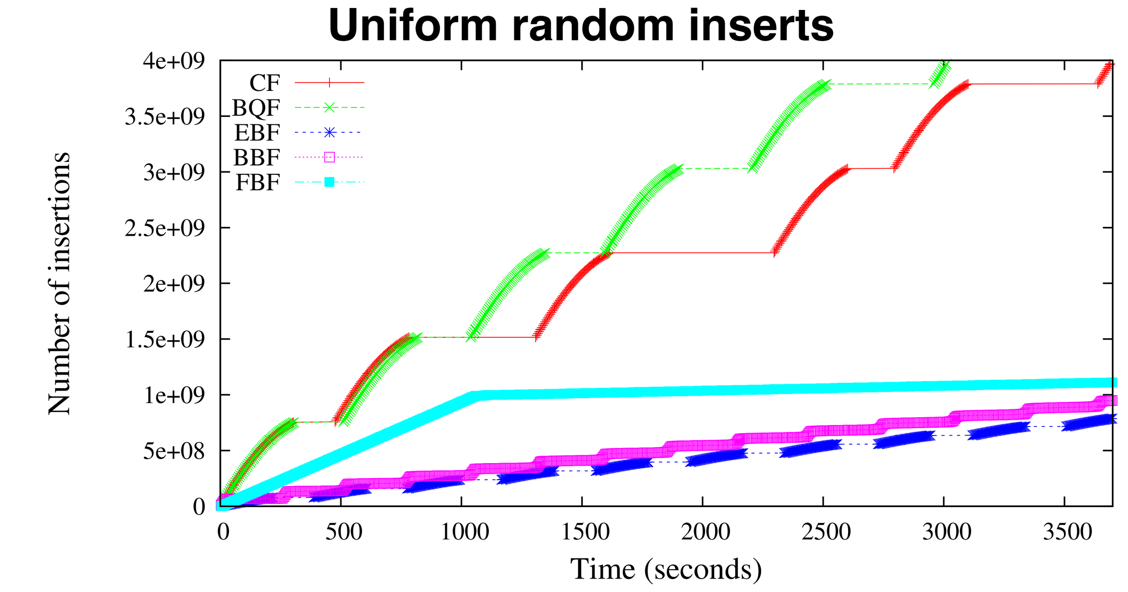
Uniform random lookups



- QF matches BF performance until about 65% occupancy.
- QF performance degrades as its occupancy increases and clusters get larger.
- BF performance degrades as the density of 1-bits increases.

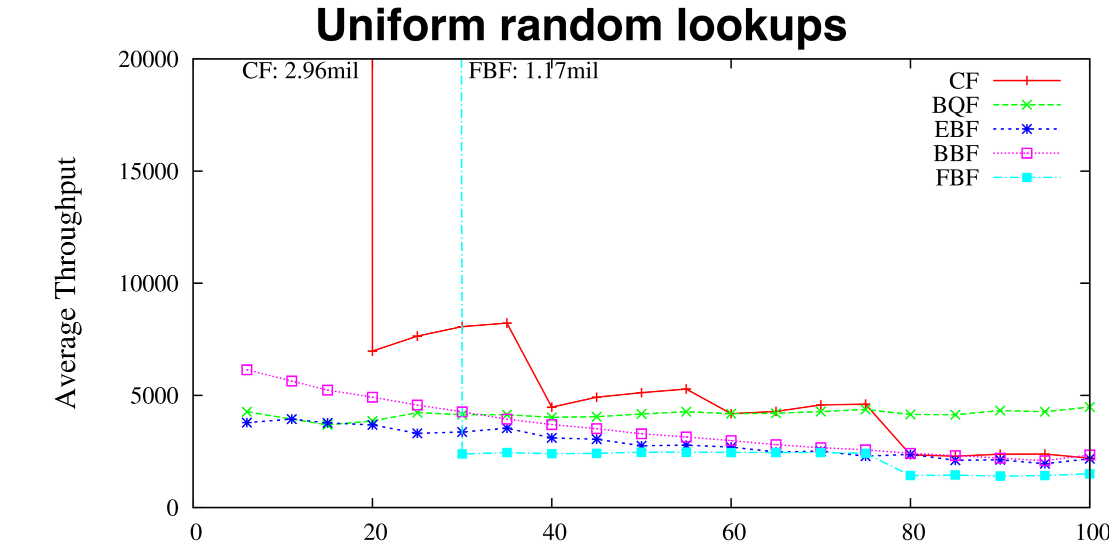
On-SSD with RAM-to-SSD ratio of 1:4

Uniform random inserts



- BQF and CF insert at least 4 times faster than all other data structures.
- The staircase pattern of the CF is due to merges.
- The stalls in the BQF are due to flushing of the buffer.

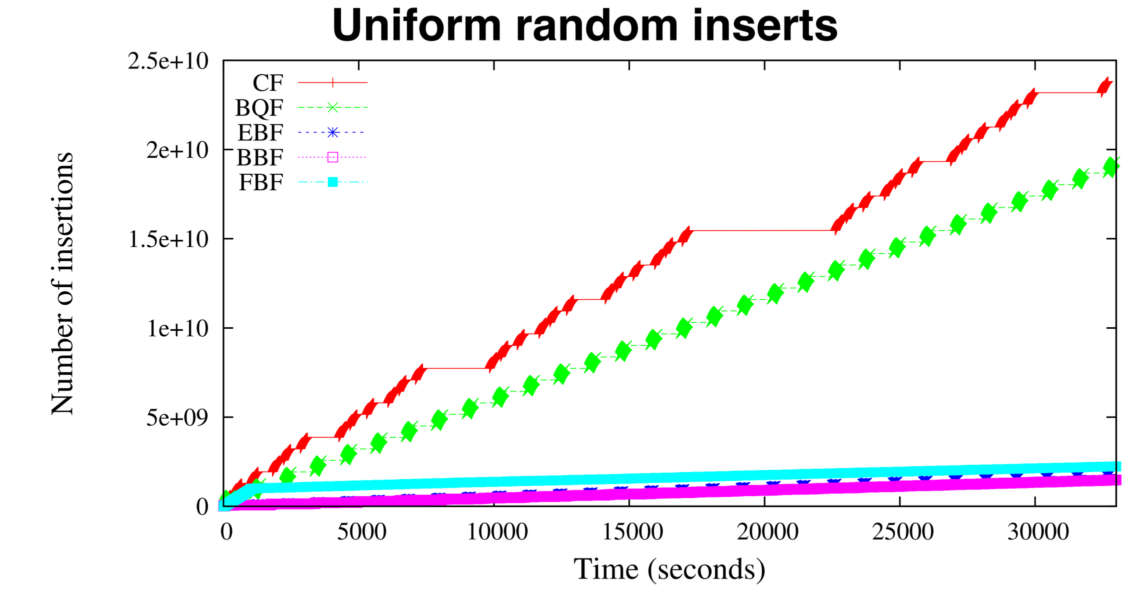
Uniform random lookups



- BQF is at least twice as fast as all other data structures.
- CF performance depends on the number of levels at a particular stage of occupancy.
- CF performance is comparable to EBF and BBF, and almost 50% higher than FBF.

On-SSD with RAM-to-SSD ratio of 1:24

Uniform random inserts



- CF and BQF insert over 23 billion records in less than 30,000 seconds. By that time, all other data structures complete less than 10% of the experiment.
- As the RAM-to-SSD ratio increases CF outperforms BQF.
- CF and BQF perform 1,940 and 3,600 uniform random lookups per second respectively. We cannot compare against EBF, BBF and FBF because they were not able to complete the large experiment.

Conclusions

- BQF and CF
  - Offer much faster inserts than recently proposed external-memory AMQ data structures and comparable lookups.
  - Are a particular good fit for decoupled workloads and write-optimized databases.
- The choice of CF versus BQF depends on the ratio of inserts to lookups in a particular workload.
- QF offers similar performance to BF but with better data locality and additional functionality.

Stony Brook University · RUTGERS · MIT Massachusetts Institute of Technology · Tokutek

Acknowledgements: This research was supported in part by DOE Grant DE-FG02-08ER25853, NSF Grants CCF-0540897, CNS-0627645, CCF-0634793, CCF-0937829, CCF-0937833, CCF-0937854, CCF-0937860, and CCF-0937822, and Politécnico Grancolombiano.

References

1. A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. Commun. ACM, 31:1116–1127, 1988.
2. M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J. Nelson. Cache-oblivious streaming B-trees. In SPAA, pages 81–92, 2007.
3. B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. Commun. ACM, 13(7):422–426, 1970.
4. M. Canim, G. A. Mihaila, B. Bhattacharhee, C. A. Lang, and K. A. Ross. Buffered Bloom filters on solid state storage. In VLDB ADMS Workshop, 2010.
5. J. G. Cleary. Compact hash tables using bidirectional linear probing. IEEE Transactions on Computing, 33(9):828–834, 1984.
6. B. Debnath, S. Sengupta, J. Li, D. Lilja, and D. Du. Bloomflash: Bloom filter on flash-based storage. In ICDCS, pages 635–644, 2011.
7. D. E. Knuth. The Art of Computer Programming: Sorting and Searching, volume 3. Addison Wesley, 1973.
8. G. Lu, B. Debnath, and D. H. C. Du. A forest-structured bloom filter with flash memory. In MSST, pages 1–6, 2011.