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

M.A. Bender, M. Farach-Colton, R. Johnson, R. Krane, 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:
  • INSERT(x).
  • 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].
• Elevator Bloom Filter (EBF) [4].
• Buffering [4, 6, 8].
• Hash localization [4, 6].
• Forest structured Bloom Filter (FFB) [8].

Quotient Filter (QF)
• Is a cache-friendly AMQ data structure.
• Maintains a p-bit fingerprint, \( f \), for each element in an open hash table with \( m = 2^p \) buckets using a technique called quotenting [7, Sec. 6.4 ex. 13].
• The fingerprint is partitioned into its \( r \) least significant bits, \( f_{rs} \), and its \( q = p - r \) most significant bits, \( f_q \).
• \( f_q \) is stored in bucket \( f_{rs} \).
• Compactly stores the hash table in an array of \( (r + 3) \)-bit items using linear probing as in [9]. 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.

Experimantal Results

In-RAM
Uniform random inserts

On-SSD with RAM-to-SSD ratio of 1:4
Uniform random inserts

On-SSD with RAM-to-SSD ratio of 1:24
Uniform random inserts

BOF 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 BOF are due to flushing of the buffer.

Conclusions

• BOF 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 BOF depends on the ratio of insert:lookup in a particular workload.
  • QF offers similar performance to BF but with better data locality and additional functionality.

References