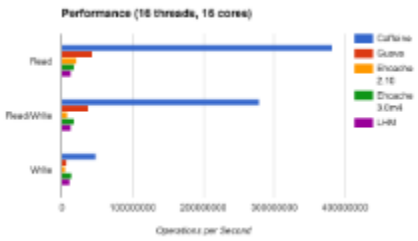
highscalability.com

Design of a Modern Cache - High Scalability -

8-10 minutes



This is a guest post by Benjamin Manes, who did engineery things for Google and is now doing engineery things for a new load documentation startup, LoadDocs.

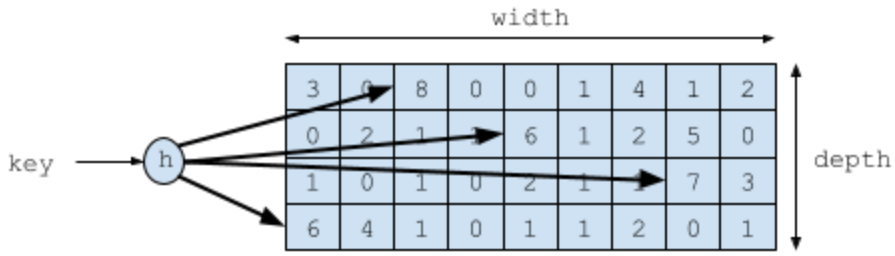
Caching is a common approach for improving performance, yet most implementations use strictly classical techniques. In this article we will explore the modern methods used by Caffeine, an open-source Java caching library, that **yield high hit rates and excellent concurrency**. These ideas can be translated to your favorite language and hopefully some readers will be inspired to do just that.

Eviction Policy

A cache's **eviction policy tries to predict which entries are most likely to be used again** in the near future, thereby maximizing the hit ratio. The Least Recently Used (LRU) policy is

perhaps the most popular due to its simplicity, good runtime performance, and a decent hit rate in common workloads. Its ability to predict the future is limited to the history of the entries residing in the cache, preferring to give the last access the highest priority by guessing that it is the most likely to be reused again soon.

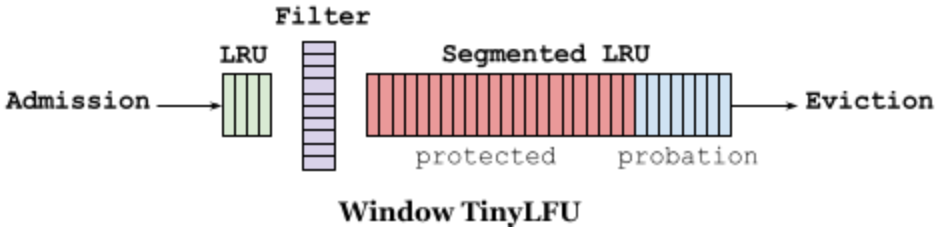
Modern caches extend the usage history to include the recent past and give preference to entries based on recency and frequency. One approach for retaining history is to use a popularity sketch (a compact, probabilistic data structure) to identify the “heavy hitters” in a large stream of events. Take for example CountMin Sketch, which **uses a matrix of counters and multiple hash functions**. The addition of an entry increments a counter in each row and the frequency is estimated by taking the minimum value observed. This approach lets us tradeoff between space, efficiency, and the error rate due to collisions by adjusting the matrix’s width and depth.



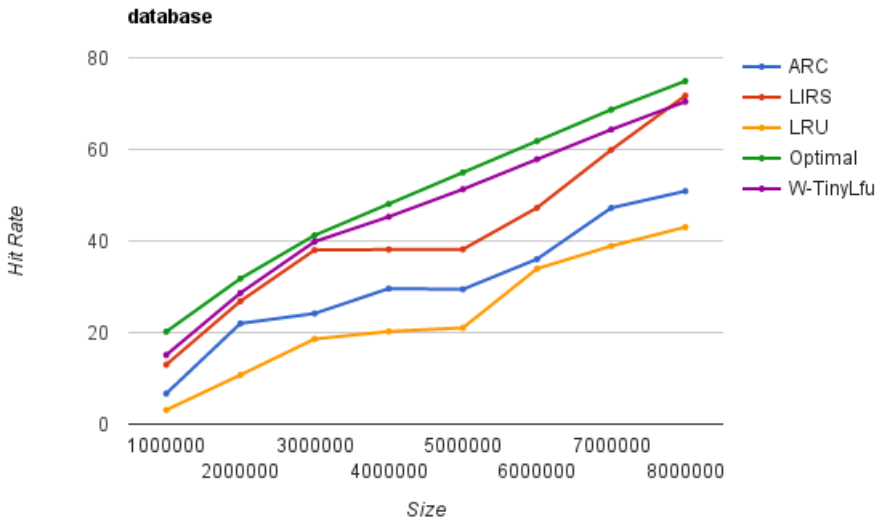
CountMin Sketch

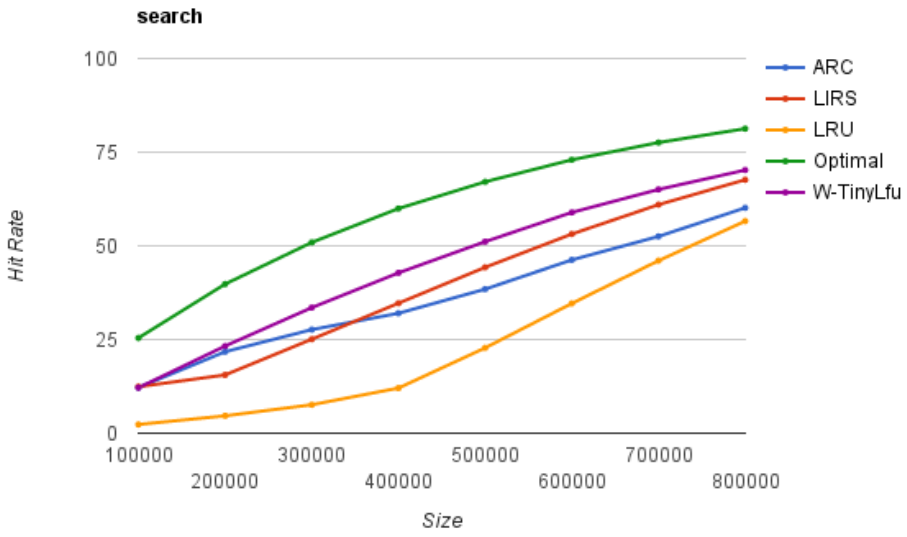
Window TinyLFU (W-TinyLFU) **uses the sketch as a filter, admitting a new entry** if it has a higher frequency than the entry that would have to be evicted to make room for it. Instead of filtering immediately, an admission window gives an entry a chance to build up its popularity. This avoids consecutive misses, especially in cases like sparse bursts where an entry may not be

deemed suitable for long-term retention. To keep the history fresh an aging process is performed periodically or incrementally to halve all of the counters.



W-TinyLFU uses the Segmented LRU (SLRU) policy for long term retention. An entry starts in the probationary segment and on a subsequent access it is promoted to the protected segment (capped at 80% capacity). When the protected segment is full it evicts into the probationary segment, which may trigger a probationary entry to be discarded. This ensures that entries with a small reuse interval (the hottest) are retained and those that are less often reused (the coldest) become eligible for eviction.





As the database and search traces show, there is a lot of opportunity to improve upon LRU by taking into account recency and frequency. More advanced policies such as ARC, LIRS, and W-TinyLFU narrow the gap to provide a near optimal hit rate. For additional workloads see the research papers and try our simulator if you have your own traces to experiment with.

Expiration Policy

Expiration is often implemented as variable per entry and expired entries are evicted lazily due to a capacity constraint. This pollutes the cache with dead items, so sometimes a scavenger thread is used to periodically sweep the cache and reclaim free space. This strategy tends to work better than ordering entries by their expiration time on a O(lg n) priority queue due to hiding the cost from the user instead of incurring a penalty on every read or write operation.

Caffeine takes a different approach by **observing that most often a fixed duration is preferred**. This constraint allows for organizing entries on O(1) time ordered queues. A time to live

duration is a write order queue and a time to idle duration is an access order queue. The cache can reuse the eviction policy's queues and the concurrency mechanism described below, so that expired entries are discarded during the cache's maintenance phase.

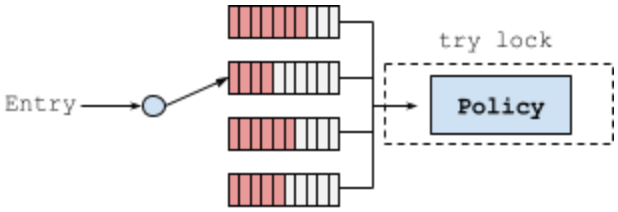
Concurrency

Concurrent access to a cache is viewed as a difficult problem because in **most policies every access is a write to some shared state**. The traditional solution is to guard the cache with a single lock. This might then be improved through lock striping by splitting the cache into many smaller independent regions. Unfortunately that tends to have a limited benefit due to hot entries causing some locks to be more contented than others. When contention becomes a bottleneck the next classic step has been to **update only per entry metadata** and use either a random sampling or a FIFO-based eviction policy. Those techniques can have great read performance, poor write performance, and difficulty in choosing a good victim.

An alternative is to **borrow an idea from database theory where writes are scaled by using a commit log**. Instead of mutating the data structures immediately, the **updates are written to a log and replayed in asynchronous batches**. This same idea can be applied to a cache by performing the hash table operation, recording the operation to a buffer, and scheduling the replay activity against the policy when deemed necessary. The policy is still guarded by a lock, or a try lock to be more precise, but shifts contention onto appending to the log buffers instead.

In Caffeine **separate buffers are used for cache reads and**

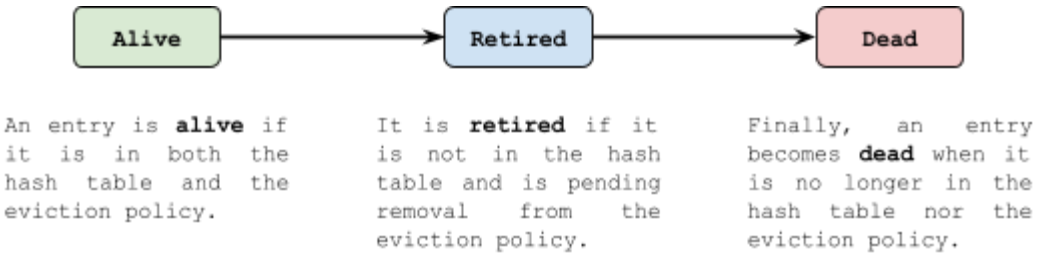
writes. An access is recorded into a **striped ring buffer** where the stripe is chosen by a thread specific hash and the number of stripes grows when contention is detected. When a ring buffer is full an asynchronous drain is scheduled and subsequent additions to that buffer are discarded until space becomes available. When the access is not recorded due to a full buffer the cached value is still returned to the caller. The loss of policy information does not have a meaningful impact because W-TinyLFU is able to identify the hot entries that we wish to retain. By using a thread-specific hash instead of the key's hash the cache avoids popular entries from causing contention by more evenly spreading out the load.



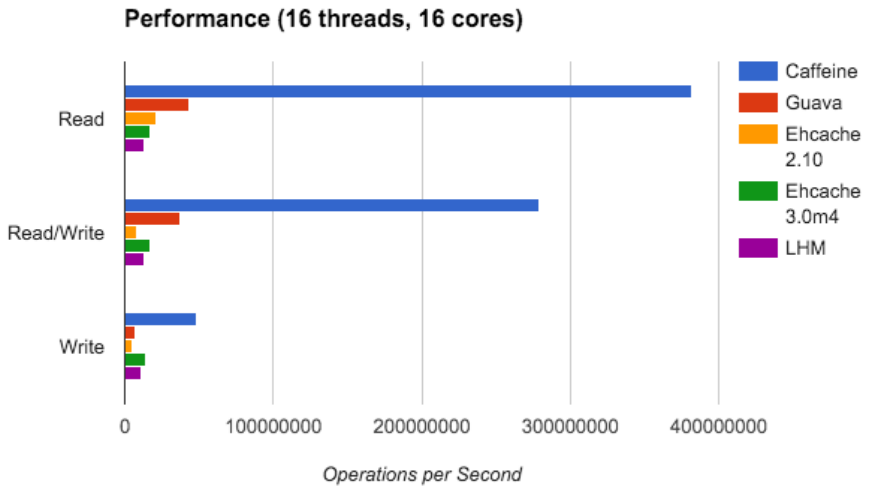
Striped Read Buffer

In the case of a write a more traditional concurrent queue is used and every change schedules an immediate drain. While data loss is unacceptable, there are still ways to optimize the write buffer. Both types of buffers are written to by multiple threads but only consumed by a single one at a given time. This multiple producer / single consumer behavior allows for simpler, more efficient algorithms to be employed.

The buffers and fine grained writes introduce a race condition where operations for an entry may be recorded out of order. An insertion, read, update, and removal can be replayed in any order and if improperly handled the policy could retain dangling references. The solution to this is a state machine defining the lifecycle of an entry.



In benchmarks the cost of the buffers is relatively cheap and scales with the underlying hash table. **Reads scale linearly with the number of CPUs** at about 33% of the hash table's throughput. Writes have a 10% penalty, but only because contention when updating the hash table is the dominant cost.



Conclusion

There are many pragmatic topics that have not been covered. This could include tricks to minimize the memory overhead, testing techniques to retain quality as complexity grows, and ways to analyze performance to determine whether a optimization is worthwhile. These are areas that practitioners must keep an eye on, because once neglected it can be difficult to restore confidence in one's own ability to manage the ensuing complexity.

The design and implementation of Caffeine is the result of

numerous insights and the hard work of many contributors. Its evolution over the years wouldn't have been possible without the help from the following people: Charles Fry, Adam Zell, Gil Einziger, Roy Friedman, Kevin Bourrillion, Bob Lee, Doug Lea, Josh Bloch, Bob Lane, Nitsan Wakart, Thomas Müeller, Dominic Tootell, Louis Wasserman, and Vladimir Blagojevic. Thanks to Nitsan Wakart, Adam Zell, Roy Friedman, and Will Chu for their feedback on drafts of this article.

Related Articles

- On HackerNews
- On Reddit
- TinyLFU - cache admission policy