Caffeine

Design of a Modern Cache

https://github.com/ben-manes/caffeine

Cache

Caffeine is a high performance, near optimal cache

Provides the familiar Guava Cache API

Lots of features (memoization, maximum size, expiration, ...)

```
LoadingCache<Key, Graph> graphs = Caffeine.newBuilder()
    .maximumSize(10_000)
    .expireAfterWrite(5, TimeUnit.MINUTES)
    .refreshAfterWrite(1, TimeUnit.MINUTES)
    .build(key -> createExpensiveGraph(key));
```

Let's focus on the data structures

Design Goals

Use O(1) algorithms for predictable performance

Optimize the system, not a single metric

Tradeoffs based on simulations

Must be correct & maintainable

APIs with low conceptual weight

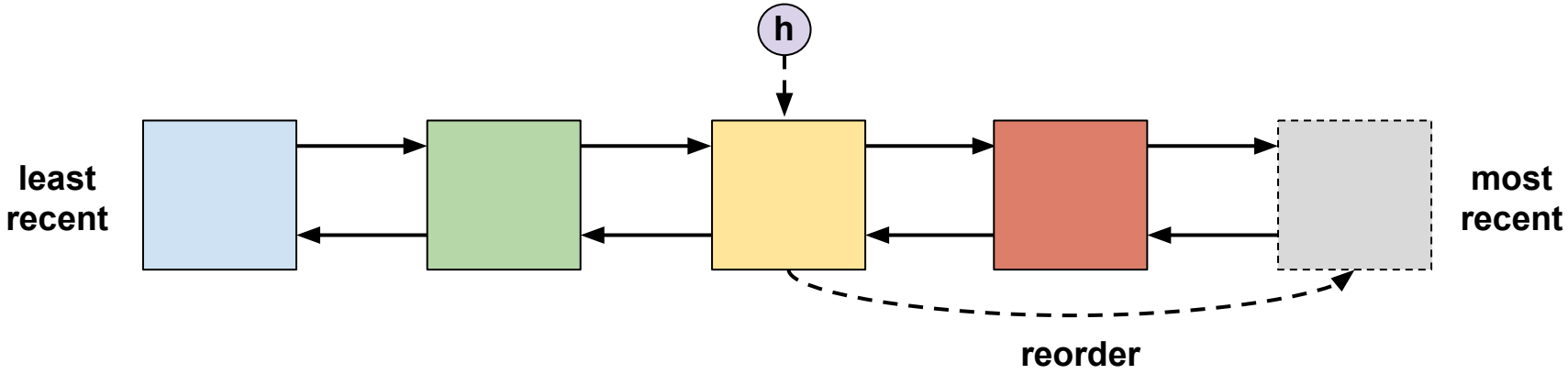
In-memory caching only

Least Recently Used

Prioritizes eviction based on access order to prefer the most recent

Suffers from cache pollution (not scan resistant, “one hit wonders”)

O(1) time for add, remove, update operations

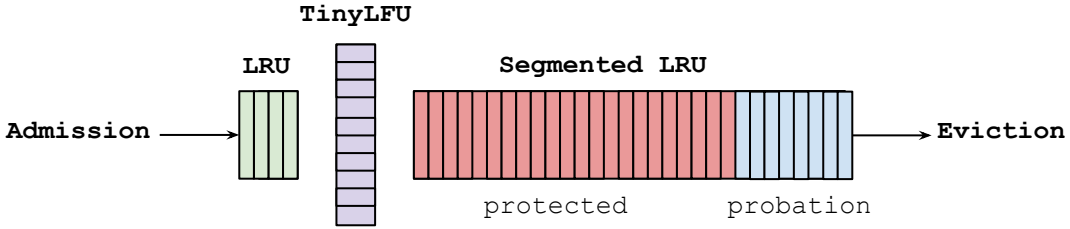


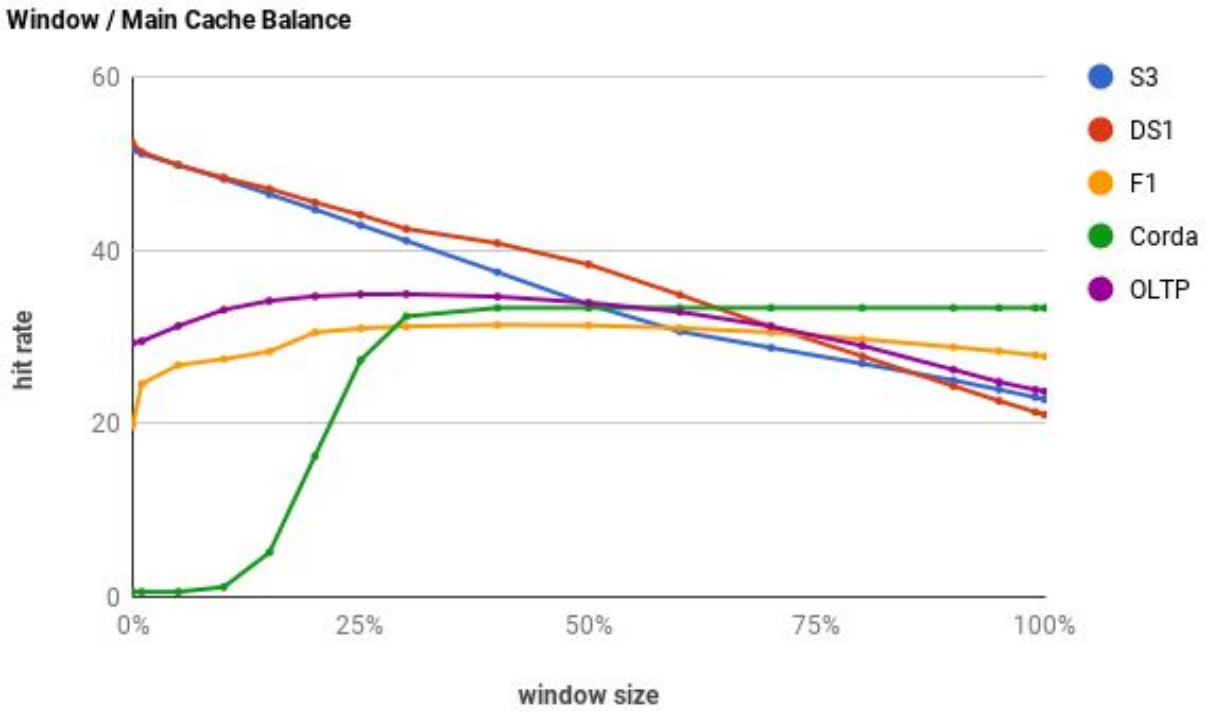
Window TinyLFU

Admit only if the estimated frequency is higher than the victim's

LRU ***before*** the filter ➜ recency-biased

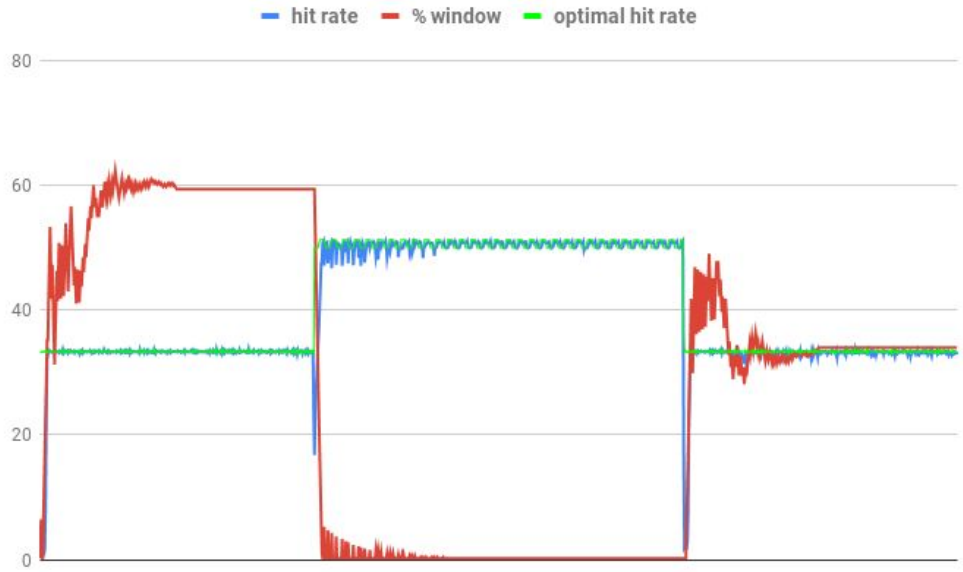
LRU ***after*** the filter ➜ frequency-biased





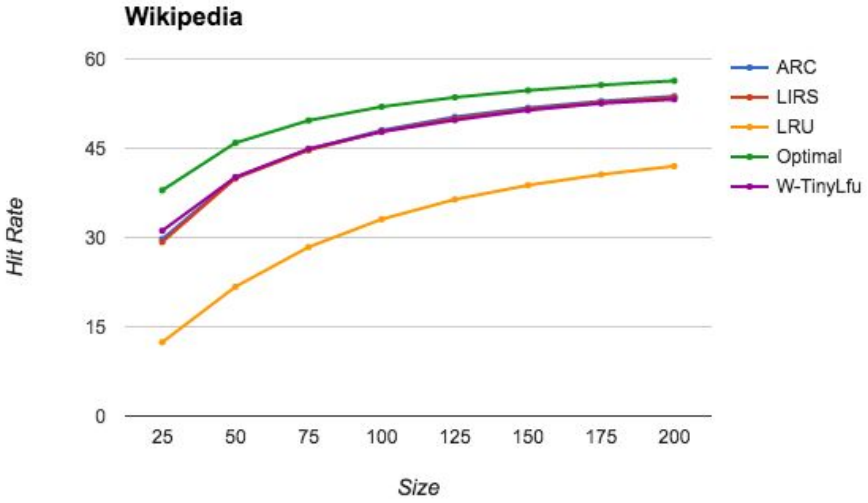
Optimize by Hill Climbing

Adaptive Window

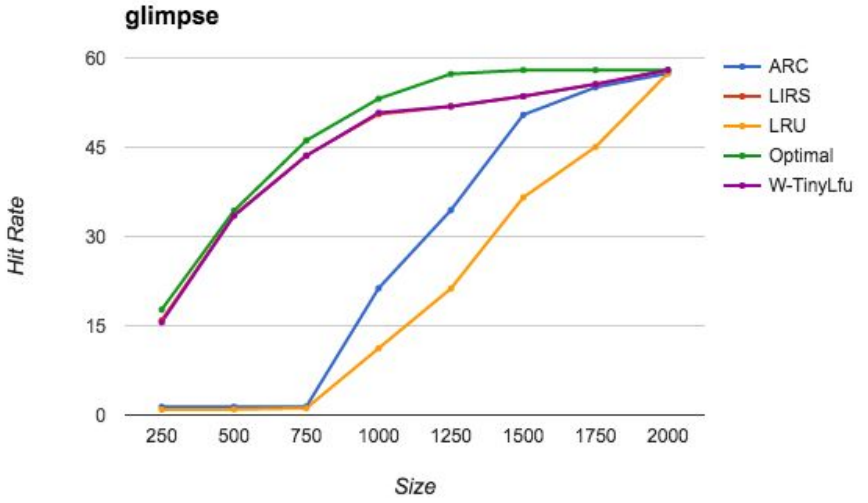


Recency ➜ Frequency ➜ Recency

Efficiency (1)

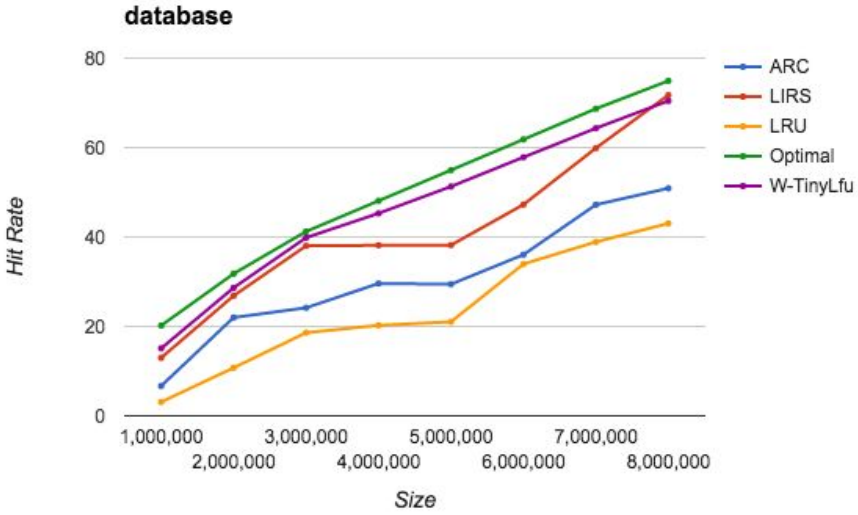


Web page Popularity

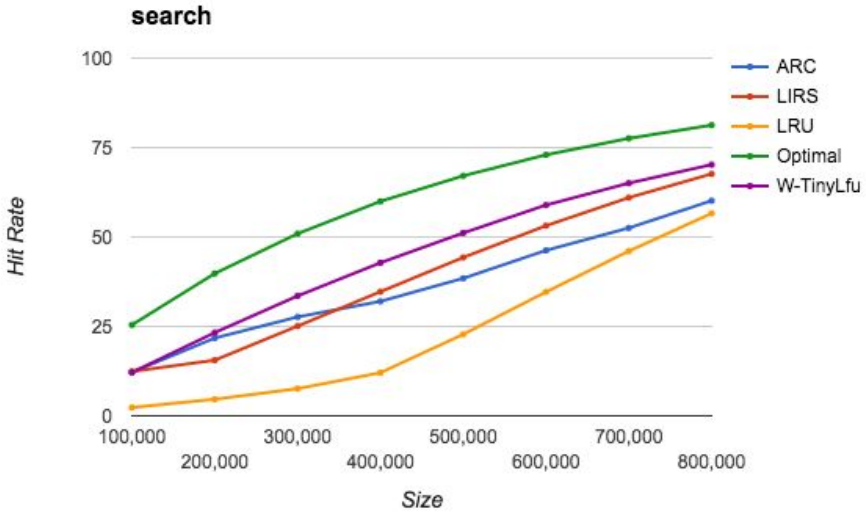


Multi-Pass Analysis

Efficiency (2)



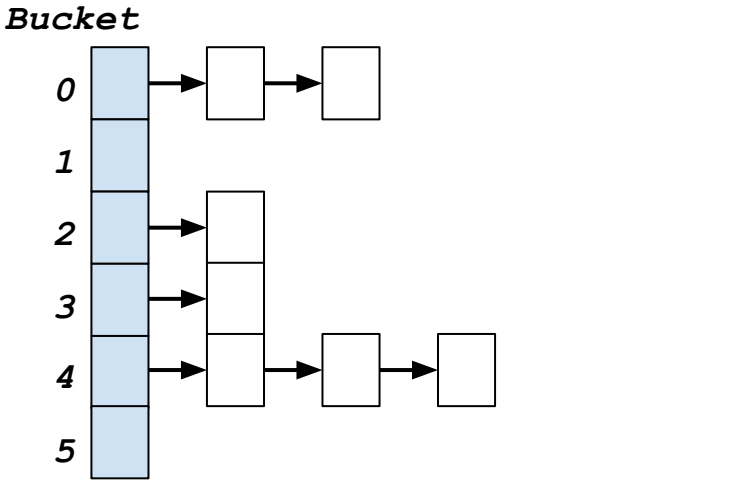
SQL Database



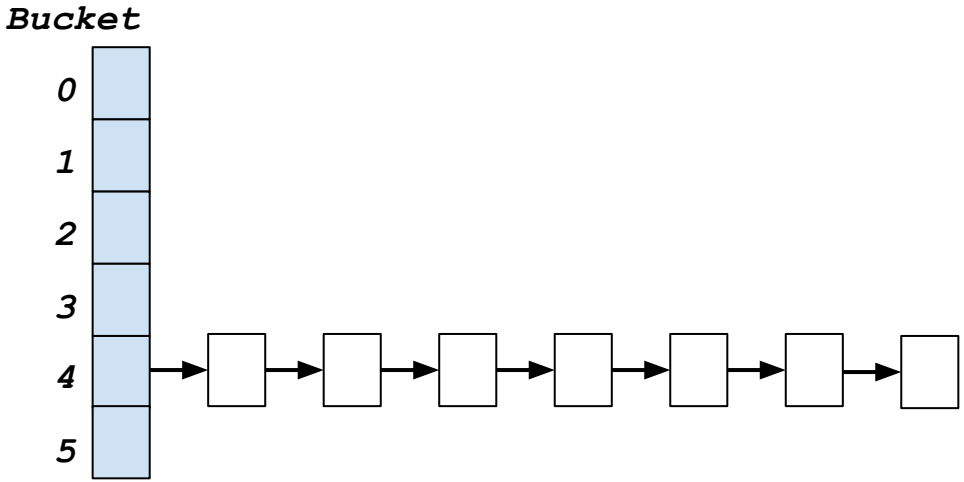
Document Search

Hash Flooding

If the attacker controls the input, he can cause worst-case performance



Normal operation of a hash table



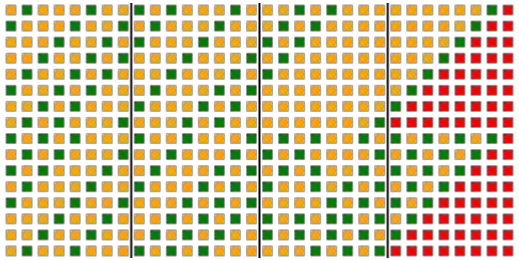
Worst-case hash table collisions

Avalanche Effect

A small input change should change in the output to be indistinguishable from random.

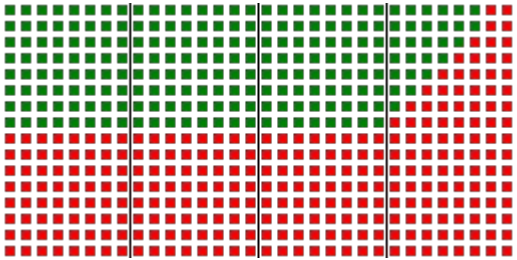
One row = one input bit
The 32 squares in each row is the influence on each hash bit by that bit.

Simple Hash



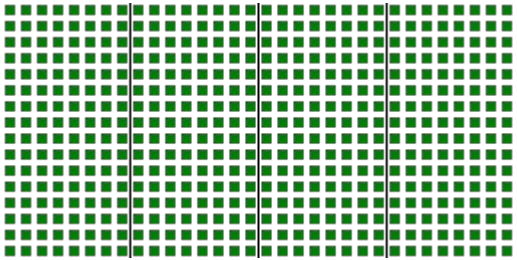
Orange is unacceptable

FNV-1 Hash



Red has no mixing at all

Jenkins96 Hash



Green is acceptable

Bret Mulvey on Hash Functions

Hash Prospecting

Good hash functions by simulated annealing!

Important criteria:

Avalanche effect: Flip one input bit ➜ each output if flipped with 50% chance

Low bias: No correlation between flipped output and input bit

github.com/skeeto/hash-prospector

```
// Thomas Mueller's hash
x = ((x >>> 16) ^ x) * 0x45d9f3b
x = ((x >>> 16) ^ x) * 0x45d9f3b
return (x >>> 16) ^ x
```

HashDoS in the wild

2MB of POST data consisting of 200,000 colliding strings ≈ 40B comparisons
Hash-flooding DoS reloaded: attacks and defenses

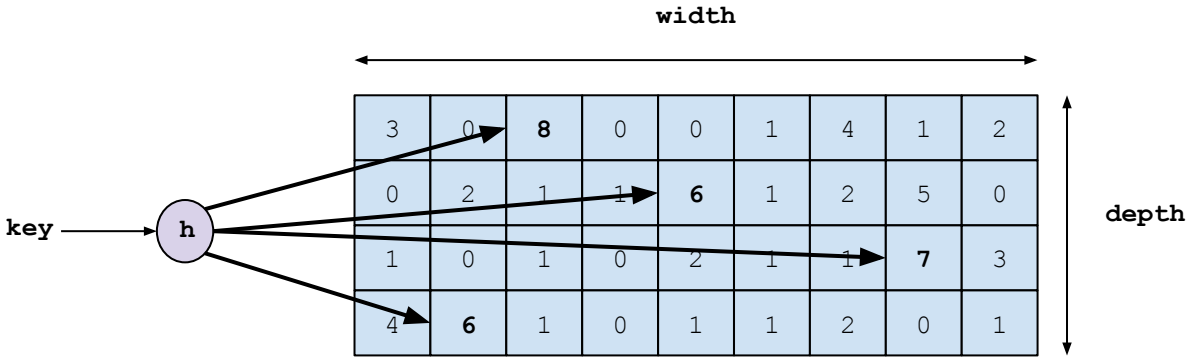
~6 kbits/s keeps one i7 core busy (1 Gbit/s ➜ 10k cores)
Efficient Denial of Service Attacks on Web Application Platforms

“A dial-up modem... can bring a dedicated server to its knees.”
Denial of Service via Algorithmic Complexity Attacks

Popularity Attack

CountMin is used to approximate heavy hitters

Attackers can exploit collisions so that the wrong entries are evicted



HashDoS mitigations

Randomization

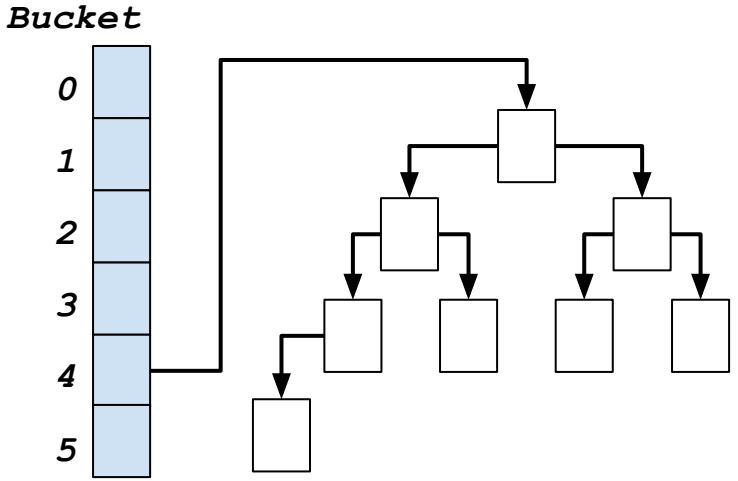
Jitter for unpredictable outcomes

Add noise and change it often, e.g. change random seed on hash table resizing

h(x) = hash(x + random seed)

Graceful degradation

Switch algorithm if needed

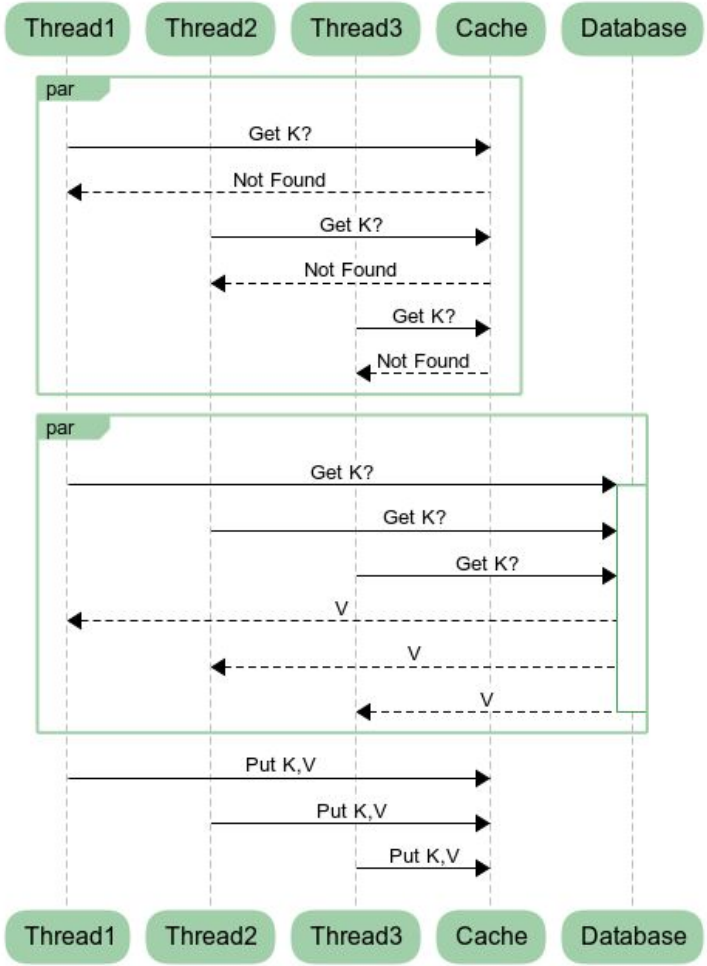


Cache Stampede

Latency spikes may be caused by concurrent loads for the same entry

Racy *get-compute-put* ➜ redundant work

Outages at Facebook, Honeycomb, and Coinbase

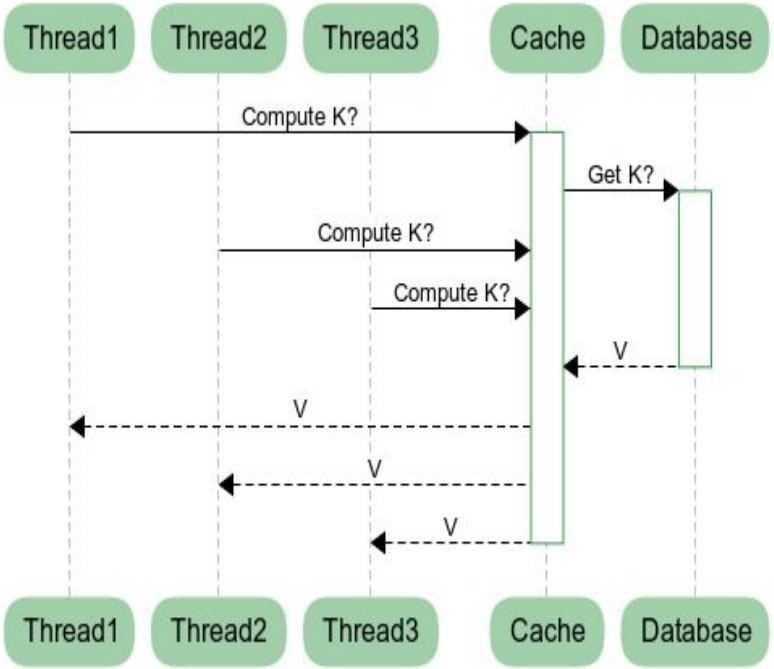


Cache Stampede

Perform the action once, atomically

```
value = cache.getIfPresent(key)
if (value == null):
  lock = locks[hash(key) % locks.length]
  with lock:
    value = cache.getIfPresent(key)
    if (value == null):
      value = compute(key)
      cache.put(key, value)
return value
```

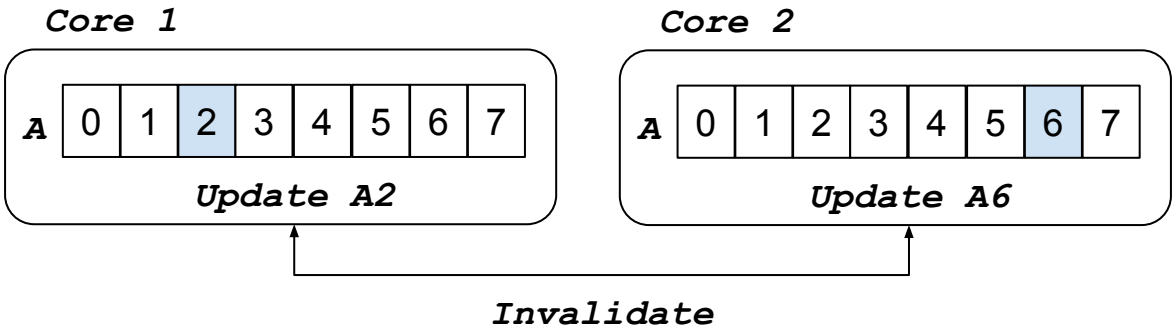
Doordash: p99 latency reduced by 87%



False Sharing

CPU caches are split into cache lines, each holding a 64-byte contiguous block

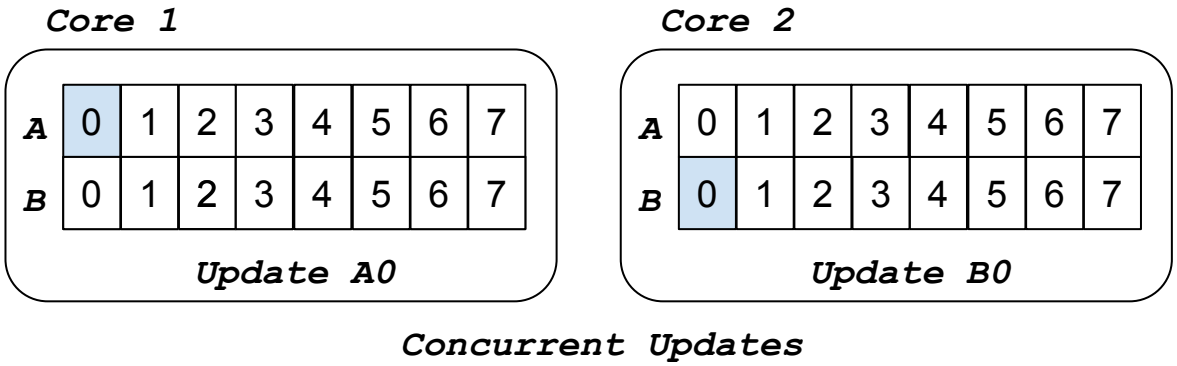
May cause hardware-level contention due to cache invalidation

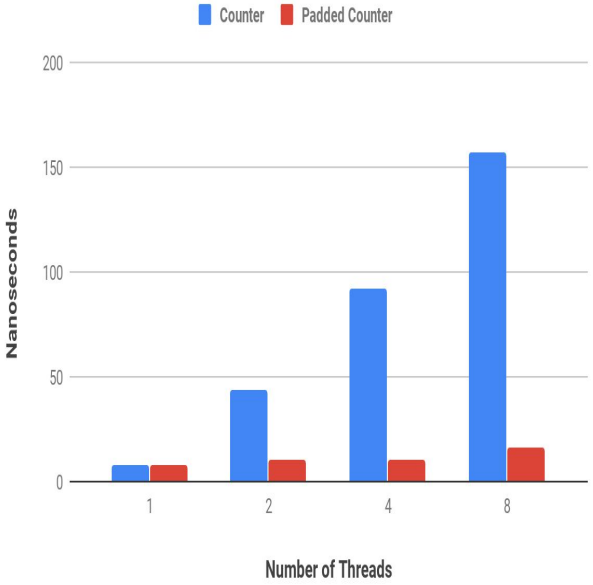


False Sharing

Pad hot variables to update independently

Example: Statistics like the hit & miss counters



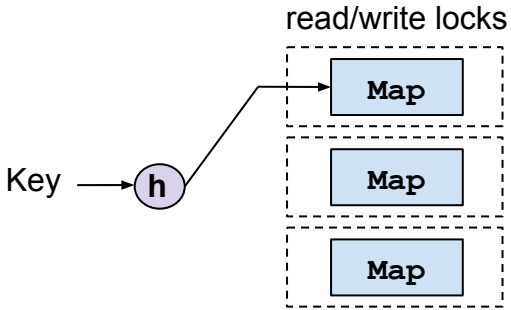


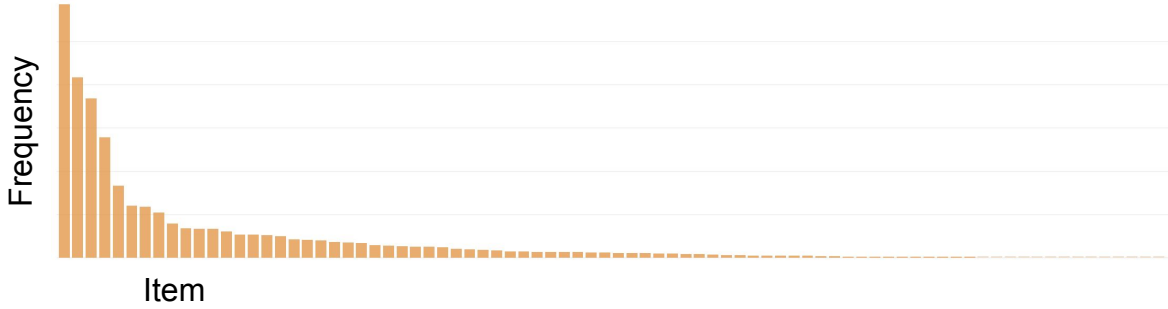
Concurrent Hash Tables

Caching may degrade performance due to synchronization!

Use read/write locks and many sub-maps?

Zipf's law causes a some locks to be used more often





Java’s ConcurrentHashMap

Lock-free reads

Locks the bin's head node during a write

Concurrently resizes the table based on the *load factor*

ReservationNodes are used as placeholders while computing values

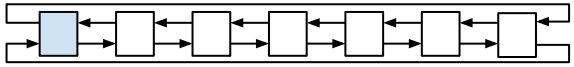
Concurrent Eviction

Linked List

Access: Move to the tail

Evict: Discard the head

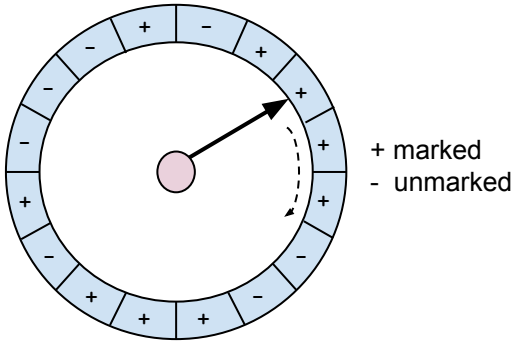
Not thread safe!



Clock

Access: Mark the entry

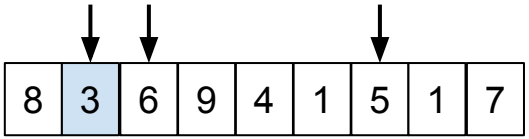
Evict: Sweep, reset, and discard if unmarked



Sampled

Access: Increment the entry's utility counter

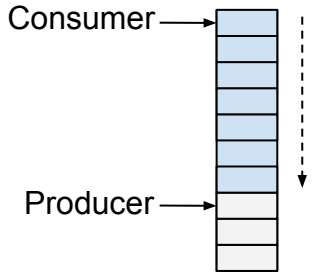
Evict: Discard the entry with the lowest utility value in a uniform distribution



Log-based Concurrency

Schedule the work, not the thread!

1. Apply the operation immediately to the hash table
2. Record the change into a read or write buffer
3. Replay under lock in asynchronous batches



BP-Wrapper: Make Any Replacement Algorithm (Almost) Lock Contention Free

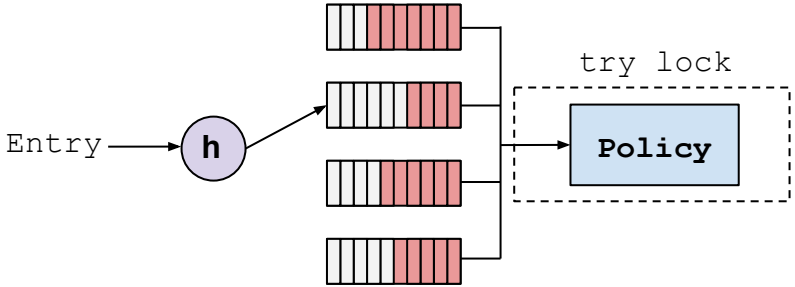
Read Buffer

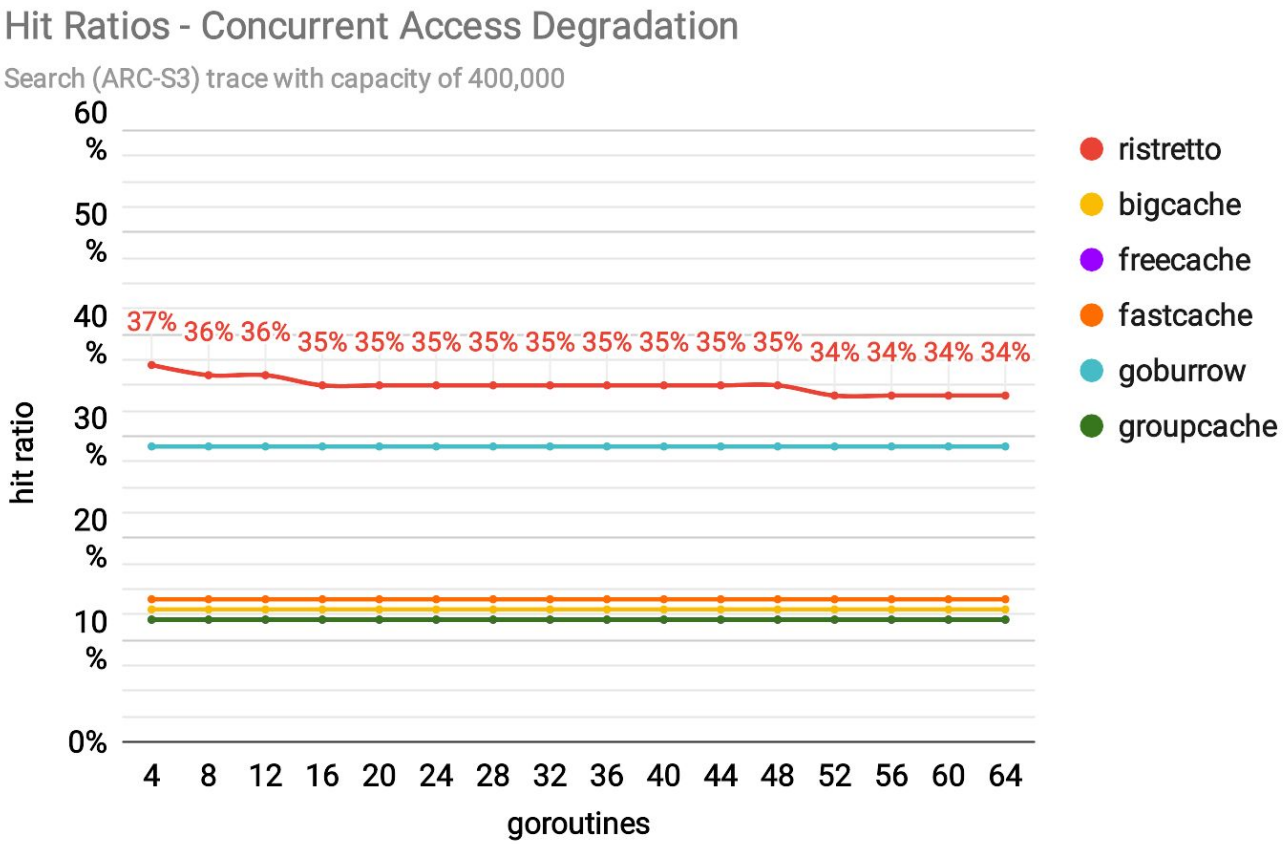
Stripe ring buffers by the thread’s id

Add buffers when contention is detected

Drop additions when full or exhausted retries

Trigger a maintenance cycle when a buffer is full




Hit Ratios - Concurrent Access Degradation
Search (ARC-S3) trace with capacity of 400,000
hit ratio
60 %
50 %
40 %
30 %
20 %
10 %
0%
37% 36% 36% 35% 35% 35% 35% 35% 35% 35% 35% 35% 34% 34% 34% 34%
4 8 12 16 20 24 28 32 36 40 44 48 52 56 60 64
goroutines
ristretto
bigcache
freecache
fastcache
goburrow
groupcache

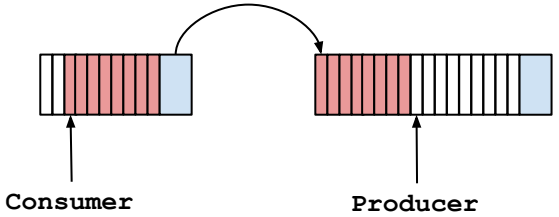
Write Buffer

Bounded ring buffer that resizes up to a maximum (via JCTools)

Back pressure when full

- Writers assist in performing the maintenance work
- Rarely occurs as requires a write rate >> replay rate

Triggers a maintenance cycle immediately

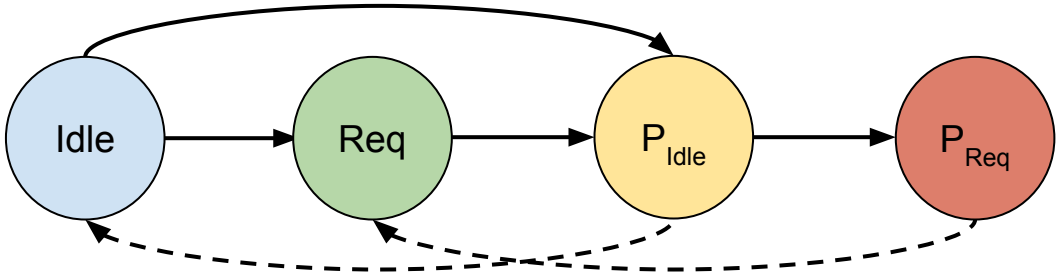


Log Replay

Schedules the async draining under a *tryLock*

Concurrently, other writes may require reprocessing

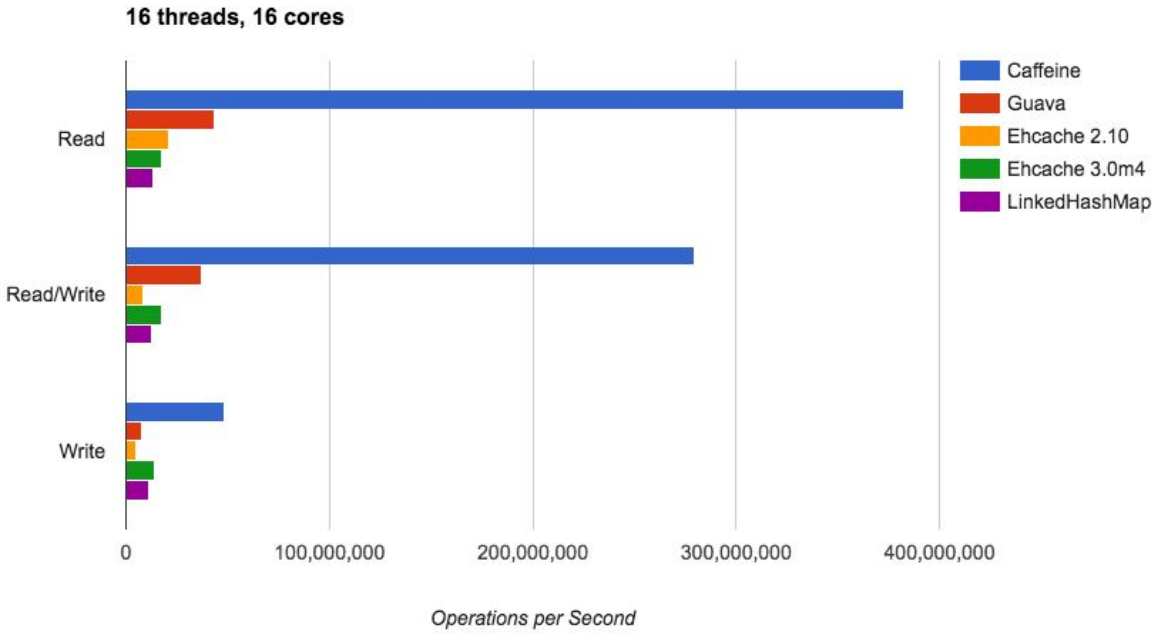
Out-of-order consistency by using per-entry states (*alive*, *retired*, or *dead*)



Replay States

- Idle
- Required
- Processing to Idle
- Processing to Required

Performance



Expiration

Lazy

Relies on size eviction

Causes cache pollution

May mitigate using a background sweeper

Fixed

Treats all entries uniformly

Simple to implement and has O(1) eviction

Limited applicability

Variable

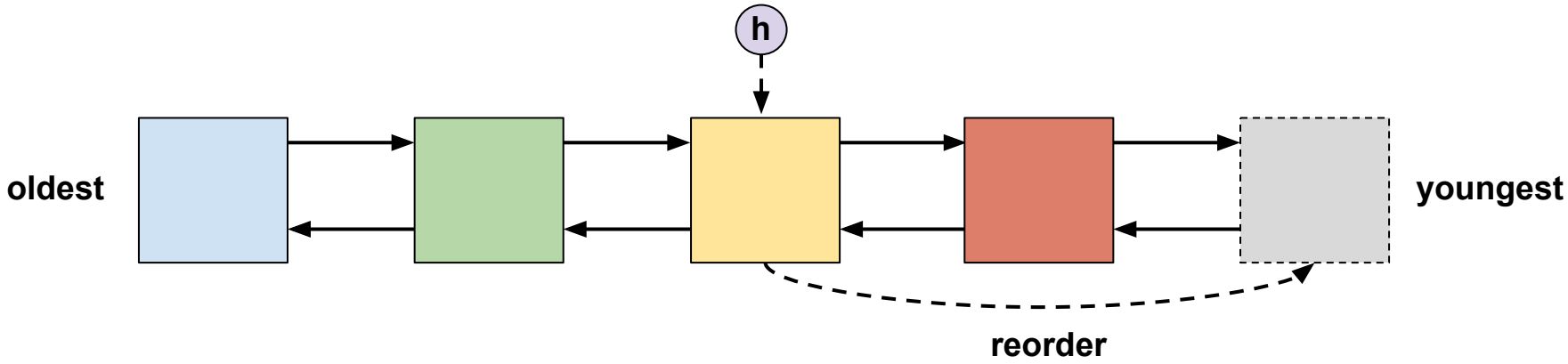
Different lifetimes per entry

Typically uses a tree, O(lg n)

May be slow at scale

Fixed Expiration

- A time-bounded Least Recently Used policy
- Time-to-Idle reorders on every access
- Time-to-Live reorders on every write



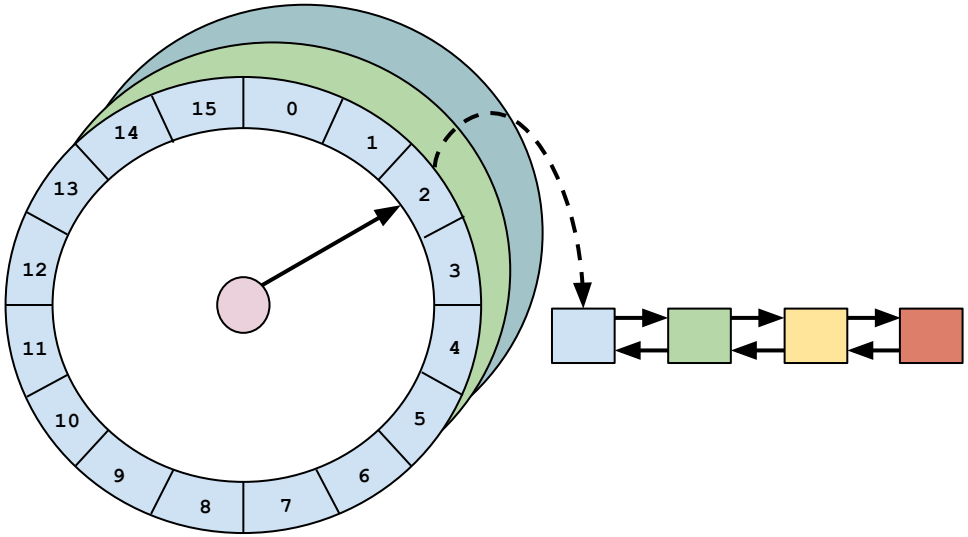
Variable Expiration

Timer Wheels provide approximate timeouts in O(1) time!

Adding to a bucket requires hashing to the coarse resolution

A clock hand “ticks” as buckets expire and can be swept

Advancing a larger wheel causes the timers to be inserted into the smaller wheels



Last Words

- Decouple and break down problems to optimize them individually
- Combine simple data structures for efficient, powerful features
- Utilize these *performance mantras*
  - Don't do it
  - Do it, but don't do it again
  - Do it cheaper
  - Do it less
  - Do it later
  - Do it when they're not looking
  - Do it concurrently