```
6.824 2015 Lecture 16: Scaling Memcache at Facebook

Scaling Memcache at Facebook, by Nishtala et al, NSDI 2013

why are we reading this paper?
  it's an experience paper, not about new ideas/techniques
  three ways to read it:
    cautionary tale of problems from not taking consistency seriously
    impressive story of super high capacity from mostly-off-the-shelf s/w
    fundamental struggle between performance and consistency
  we can argue with their design, but not their success

how do web sites scale up with growing load?
  a typical story of evolution over time:
  1. one machine, web server, application, DB
     DB stores on disk, crash recovery, transactions, SQL
     application queries DB, formats, HTML, &c
     but the load grows, your PHP application takes too much CPU time
  2. many web FEs, one shared DB
     an easy change, since web server + app already separate from storage
     FEs are stateless, all sharing (and concurrency control) via DB
     but the load grows; add more FEs; soon single DB server is bottleneck
  3. many web FEs, data sharded over cluster of DBs
     partition data by key over the DBs
       app looks at key (e.g. user), chooses the right DB
     good DB parallelism if no data is super-popular
     painful -- cross-shard transactions and queries probably don't work
       hard to partition too finely
     but DBs are slow, even for reads, why not cache read requests?
  4. many web FEs, many caches for reads, many DBs for writes
     cost-effective b/c read-heavy and memcached 10x faster than a DB
       memcached just an in-memory hash table, very simple
     complex b/c DB and memcacheds can get out of sync
     (next bottleneck will be DB writes -- hard to solve)

the big facebook infrastructure picture
  lots of users, friend lists, status, posts, likes, photos
    fresh/consistent data apparently not critical
    because humans are tolerant?
  high load: billions of operations per second
    that's 10,000x the throughput of one DB server
  multiple data centers (at least west and east coast)
  each data center -- "region":
    "real" data sharded over MySQL DBs
    memcached layer (mc)
    web servers (clients of memcached)
  each data center's DBs contain full replica
  west coast is master, others are slaves via MySQL async log replication

how do FB apps use mc?
  read:
    v = get(k) (computes hash(k) to choose mc server)
    if v is nil {
      v = fetch from DB
      set(k, v)
    }
  write:
    v = new value
    send k,v to DB
    delete(k)
```

```
  application determines relationship of mc to DB
    mc doesn't know anything about DB
  FB uses mc as a "look-aside" cache
    real data is in the DB
    cached value (if any) should be same as DB

what does FB store in mc?
  paper does not say
  maybe userID -> name; userID -> friend list; postID -> text; URL -> likes
  basically copies of data from DB

paper lessons:
  look-aside is much trickier than it looks -- consistency
    paper is trying to integrate mutually-oblivious storage layers
  cache is critical:
    not really about reducing user-visible delay
    mostly about surviving huge load!
    cache misses and failures can create intolerable DB load
  they can tolerate modest staleness: no freshness guarantee
  stale data nevertheless a big headache
    want to avoid unbounded staleness (e.g. missing a delete() entirely)
    want read-your-own-writes
    each performance fix brings a new source of staleness
  huge "fan-out" => parallel fetch, in-cast congestion

let's talk about performance first
  majority of paper is about avoiding stale data
  but staleness only arose from performance design

performance comes from parallel get()s by many mc servers
  driven by parallel processing of HTTP requests by many web servers
  two basic parallel strategies for storage: partition vs replication

will partition or replication yield most mc throughput?
  partition: server i, key k -> mc server hash(k)
  replicate: server i, key k -> mc server hash(i)
  partition is more memory efficient (one copy of each k/v)
  partition works well if no key is very popular
  partition forces each web server to talk to many mc servers (overhead)
  replication works better if a few keys are very popular

performance and regions (Section 5)

Q: what is the point of regions -- multiple complete replicas?
   lower RTT to users (east coast, west coast)
   parallel reads of popular data due to replication
   (note DB replicas help only read performance, no write performance)
   maybe hot replica for main site failure?

Q: why not partition users over regions?
   i.e. why not east-coast users' data in east-coast region, &c
   social net -> not much locality
   very different from e.g. e-mail

Q: why OK performance despite all writes forced to go to the master region?
   writes would need to be sent to all regions anyway -- replicas
   users probably wait for round-trip to update DB in master region
     only 100ms, not so bad
   users do not wait for all effects of writes to finish
     i.e. for all stale cached values to be deleted
```

```
  performance within a region (Section 4)

  multiple mc clusters *within* each region
    cluster == complete set of mc cache servers
      i.e. a replica, at least of cached data

  why multiple clusters per region?
    why not add more and more mc servers to a single cluster?
    1. adding mc servers to cluster doesn't help single popular keys
       replicating (one copy per cluster) does help
    2. more mcs in cluster -> each client req talks to more servers
       and more in-cast congestion at requesting web servers
       client requests fetch 20 to 500 keys! over many mc servers
       MUST request them in parallel (otherwise total latency too large)
       so all replies come back at the same time
       network switches, NIC run out of buffers
    3. hard to build network for single big cluster
       uniform client/server access
       so cross-section b/w must be large -- expensive
       two clusters -> 1/2 the cross-section b/w

  but -- replicating is a waste of RAM for less-popular items
    "regional pool" shared by all clusters
    unpopular objects (no need for many copies)
    decided by *type* of object
    frees RAM to replicate more popular objects

  bringing up new mc cluster was a serious performance problem
    new cluster has 0% hit rate
    if clients use it, will generate big spike in DB load
      if ordinarily 1% miss rate, and (let's say) 2 clusters,
        adding "cold" third cluster will causes misses for 33% of ops.
      i.e. 30x spike in DB load!
    thus the clients of new cluster first get() from existing cluster (4.3)
      and set() into new cluster
      basically lazy copy of existing cluster to new cluster
    better 2x load on existing cluster than 30x load on DB

  important practical networking problems:
    n^2 TCP connections is too much state
      thus UDP for client get()s
    UDP is not reliable or ordered
      thus TCP for client set()s
      and mcrouter to reduce n in n^2
    small request per packet is not efficient (for TCP or UDP)
      per-packet overhead (interrupt &c) is too high
      thus mcrouter batches many requests into each packet

  mc server failure?
    can't have DB servers handle the misses -- too much load
    can't shift load to one other mc server -- too much
    can't re-partition all data -- time consuming
    Gutter -- pool of idle servers, clients only use after mc server fails

  The Question:
    why don't clients send invalidates to Gutter servers?
    my guess: would double delete() traffic
      and send too many delete()s to small gutter pool
      since any key might be in the gutter pool

  thundering herd
```

```
  one client updates DB and delete()s a key
  lots of clients get() but miss
    they all fetch from DB
    they all set()
  not good: needless DB load
  mc gives just the first missing client a "lease"
    lease = permission to refresh from DB
    mc tells others "try get() again in a few milliseconds"
  effect: only one client reads the DB and does set()
    others re-try get() later and hopefully hit

let's talk about consistency now

the big truth
  hard to get both consistency (== freshness) and performance
  performance for reads = many copies
  many copies = hard to keep them equal

what is their consistency goal?
  *not* read sees latest write
    since not guaranteed across clusters
  more like "not more than a few seconds stale"
    i.e. eventual
  *and* writers see their own writes
    read-your-own-writes is a big driving force

first, how are DB replicas kept consistent across regions?
  one region is master
  master DBs distribute log of updates to DBs in slave regions
  slave DBs apply
  slave DBs are complete replicas (not caches)
  DB replication delay can be considerable (many seconds)

how do we feel about the consistency of the DB replication scheme?
  good: eventual consistency, b/c single ordered write stream
  bad: longish replication delay -> stale reads

how do they keep mc content consistent w/ DB content?
  1. DBs send invalidates (delete()s) to all mc servers that might cache
  2. writing client also invalidates mc in local cluster
     for read-your-writes

why did they have consistency problems in mc?
  client code to copy DB to mc wasn't atomic:
    1. writes: DB update ... mc delete()
    2. read miss: DB read ... mc set()
  so *concurrent* clients had races

what were the races and fixes?

Race 1:
  k not in cache
  C1 get(k), misses
  C1 v = read k from DB
    C2 updates k in DB
    C2 and DB delete(k) -- does nothing
  C1 set(k, v)
  now mc has stale data, delete(k) has already happened
  will stay stale indefinitely, until key is next written
  solved with leases -- C1 gets a lease, but C2's delete() invalidates lease,
    so mc ignores C1's set
```

```
    key still missing, so next reader will refresh it from DB

Race 2:
  during cold cluster warm-up
  remember clients try get() in warm cluster, copy to cold cluster
  k starts with value v1
  C1 updates k to v2 in DB
  C1 delete(k) -- in cold cluster
  C2 get(k), miss -- in cold cluster
  C2 v1 = get(k) from warm cluster, hits
  C2 set(k, v1) into cold cluster
  now mc has stale v1, but delete() has already happened
    will stay stale indefinitely, until key is next written
  solved with two-second hold-off, just used on cold clusters
    after C1 delete(), cold ignores set()s for two seconds
    by then, delete() will propagate via DB to warm cluster

Race 3:
  k starts with value v1
  C1 is in a slave region
  C1 updates k=v2 in master DB
  C1 delete(k) -- local region
  C1 get(k), miss
  C1 read local DB  -- sees v1, not v2!
  later, v2 arrives from master DB
  solved by "remote mark"
    C1 delete() marks key "remote"
    get()/miss yields "remote"
      tells C1 to read from *master* region
    "remote" cleared when new data arrives from master region

Q: aren't all these problems caused by clients copying DB data to mc?
   why not instead have DB send new values to mc, so clients only read mc?
     then there would be no racing client updates &c, just ordered writes
A:
  1. DB doesn't generally know how to compute values for mc
     generally client app code computes them from DB results,
       i.e. mc content is often not simply a literal DB record
  2. would increase read-your-own writes delay
  3. DB doesn't know what's cached, would end up sending lots
     of values for keys that aren't cached

PNUTS does take this alternate approach of master-updates-all-copies

FB/mc lessons for storage system designers?
  cache is vital to throughput survival, not just a latency tweak
  need flexible tools for controlling partition vs replication
  need better ideas for integrating storage layers with consistency
```