```
6.824 2021 Lecture 17: 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 about not taking consistency seriously from the start
    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 cope as they get more users?
  a typical story of evolution over time:
  1. single machine w/ web server + application + DB
     DB provides persistent storage, crash recovery, transactions, SQL
     application queries DB, formats HTML, &c
     but: as load grows, 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
       stateless -> any FE can serve any request, no harm from FE crash
     but: as load grows, need 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
     fragile b/c cache misses can easily overload the DB
     (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 not critical
      humans are tolerant
  high load: billions of operations per second
    that's 10,000x the throughput of one DB server
      ~100,000 simple queries/s for mysql
      ~1,000s transaction/s
      ~1000,000 get/puts/s  for memcached
  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 primary, others are secondary replicas via MySQL async log replication

how do FB apps use mc? Figure 1.
  FB uses mc as a "look-aside" cache
    real data is in the DB
    cached value (if any) should be same as DB
  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

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

paper lessons:
  look-aside caching is 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 shielding DB from huge overload!
  human users can tolerate modest read staleness
  stale reads nevertheless potentially a big headache
    want to avoid unbounded staleness (e.g. missing a delete() entirely)
    want read-your-own-writes
    more caches -> more sources 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 parallelism due to many servers
  many active users, many web servers (clients)
  two basic parallel strategies for storage: partition vs replication

will partition or replication yield most mc throughput?
  partition: divide keys over mc servers
  replicate: divide clients over mc servers
  partition:
    + more memory-efficient (one copy of each k/v)
    + works well if no key is very popular
    - each web server must talk to many mc servers (overhead)
  replication:
    + good if a few keys are very popular
    + fewer TCP connections
    - less total data can be cached

performance and regions (Section 5)

[diagram: west, db primary shards, mc servers, clients | east, db secondary shards, ...
 feed from db primary to secondary ]

Q: what is the point of regions -- multiple complete replicas?
   lower RTT to users (east coast, west coast)
   quick local reads, from local mc and DB
   (though writes are expensive: must be sent to primary)
   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
   then no need to replicate: might cut hardware costs in half!
   but: social net -> not much locality
   might work well for e.g. e-mail

Q: why OK performance despite all writes forced to go to the primary region?
   writes are much rarer than reads
   perhaps 100ms to send write to primary, not so bad for human users
```

```
    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)

[diagram: db shards, multiple clusters, each w/ mc's and clients ]

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)
  the application s/w decides what to put in regional pool
  frees RAM to replicate more popular objects

bringing up new mc cluster is a performance problem
  new cluster has 0% hit rate
  if clients use it, will generate big spike in DB load
    if ordinarily 1% miss rate,
      adding "cold" second cluster will causes misses for 50% of ops.
    i.e. 50x 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

another overload problem: 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

what if an mc server fails?
  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 mc servers, clients only use after mc server fails
  after a while, failed mc server will be replaced

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

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
  single 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

let's talk about consistency now

what is their consistency goal?
  writes go direct to primary DB, with transactions, so writes are consistent
  what about reads?
  reads do not always see the latest write
    e.g. since not guaranteed across clusters
  more like "not more than a few seconds stale"
    i.e. eventual
  *and* writers see their own writes (due to delete())
    read-your-own-writes is a big driving force

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

how do they keep mc content consistent w/ DB content?
  1. DBs send invalidates (delete()s) to all mc servers that might cache
     this is McSqueal in Figure 6
  2. writing client also invalidates mc in local cluster
     for read-your-own-writes

they ran into a number of DB-vs-mc consistency problems
  due to races when multiple clients read from DB and put() into mc
  or: there is not a single path along which updates flow in order

what were the races and fixes?

Race 1:
  k not in cache
  C1 get(k), misses
  C1 v1 = read k from DB
    C2 writes k = v2 in DB
    C2 delete(k)
  C1 set(k, v1)
  now mc has stale data, delete(k) has already happened
  will stay stale indefinitely, until k is next written
  solved with leases -- C1 gets a lease from mc, 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: on miss, 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 mc ignores set()s for two seconds
    by then, delete() will (probably) propagate via DB to warm cluster

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

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

--- references

http://cs.cmu.edu/~beckmann/publications/papers/2020.osdi.cachelib.pdf
```