```
Cooper et al, PNUTS: Yahoo!'s Hosted Data Serving Platform. VLDB 2008.
Notes from Robert Morris

PNUTS:  multi-region data replication -- facing 100ms network delays
  stronger guarantees than Dynamo
  apps control trade-off between consistency and performance

PNUTS' overall story similar to that Dynamo
  data centers ("regions") all over the world
  each region has web servers and storage
  the big point: low latency via user contacting nearest region
  for small records: user profile, messages, shopping cart, friend list, &c

design overview
  each region has all data
  each table partitioned by key over storage units

why replicas of *all* data at multiple regions?
  any user can read anything quickly
    good for data used by many users / many regions
    cross-region reads would be slow (~100ms)
  failed SU repairable from another replica

drawbacks of one copy at every region?
  updates need to be sent to every region
  local reads will probably be stale
  updates from multiple regions need to be sorted out
    keep replicas identical
    don't lose updates (e.g. read-modify-write for counter)
  uses more disk space? not a problem for small records.

how do updates work?
  app server gets web request, needs to write data in PNUTS
  need to update every region!
  why not just have app logic send update to every region?
    what if app crashes after updating only some regions?
    what if concurrent updates to same record?

PNUTS has a "record master" for each record
  master imposes order on writes for each record
  responsible storage unit executes updates one at a time per record
  tells MB to broadcast updates in order to all regions

why per-record master?
  each record has a hidden column indicating region of record master
  finer-grained than Facebook/memcache master region
  hopefully get best of both worlds:
    fast reads of local data
    fast writes b/c master is often the local region

the complete update story (some guesses):
  app wants to update some columns of a record, knows key
  1. app sends update
  2. router forwards to local SU1
  3. SU1 looks up record master for key: region R2
  4. SU1 sends update request to router in R2
  5. R2 router forwards update to local SU2
  6. SU2 sends update to local Message Broker (MB)
  7. MB stores on disk + backup MB, sends vers # to original app
  8. MB sends update to MB at every region
  9. every region updates local copy (MB, router, SU)

MB is a neat idea
  atomic: updates all replicas, or none
    rather than app server updating replicas (crash...)
  reliable: logs to disk, keeps trying, to cope with various failures
  ordered: record's replicas eventually identical even w/ multiple writers
  async: apps don't wait once write committed at MB
```



```
write order semantics
  suppose each record in a table looks like:
    Name   Where  What
    Alice  home   sleeping
  PNUTS preserves order of writes to a single record
    e.g. Alice does
      write(Alice.What, awake)
      write(Alice.Where, work)
    write RPC only returns after committed to record master's MB
    readers may see home/sleeping or home/awake or work/awake
    no-one will see work / sleeping
  BUT writes to different records do not maintain order
    e.g.
      Write(Bob.What, on duty)
      Write(Alice.What, off duty)
    readers may these in either order
      i.e. no-one on duty; or two people on duty
    if atomic hand-off important, need a single record with name of person

why is it OK for writes to take effect slowly?
  fundamental benefit: reads are then very fast, since local
    and reads usually greatly outnumber writes
  PNUTS mitigates write cost:
    application waits for MB commit but not propagation ("asynchronous")
    master likely to be local (they claim 80% of the time)
      so MB commit will often be quick
  still, eval says write takes 300ms if master is remote!
  down side: readers at non-master regions may see stale data

how stale might a non-master record be?
  depends on how quickly MB sends updates to regions
  guess: less than a second usually
    longer if network slow/flaky, or MB busy

how do apps cope with potentially stale local replica?
  sometimes stale is ok: looking at other people's profiles
  sometimes stale is not ok: shopping cart after add/delete item
  application gets to choose how consistent (section 2.2)
  read-any(k)
    read from local SU
    fast but maybe stale
  read-critical(k, required_version)
    fast local read if local SU has vers >= required_version
    otherwise slow read from master SU
    why: app knows it just wrote, wants value to reflect write
  read-latest(k)
    always read from master SU
    slow if master is remote!
    why: app needs fresh data

what if app needs to increment a counter stored in a record?
  is read-latest(k), increment, then write(k) enough?
  not if there might be concurrent updates!

test-and-set-write(version#, new value) gives you atomic update to one record
  master rejects the write if current version # != version#
  so if concurrent updates, one will lose and retry
  while(1):
    (x, ver) = read-latest(k)
    if(t-a-s-w(k, ver, x+1))
      break
  t-a-s-w is fully general for single-record atomic read-modify-write

The Question
  how does PNUTS cope with Example 1 (page 2)
  Initially Alice's mother is in Alice's ACL, so mother can see photos
  1. Alice removes her mother from ACL
  2. Alice posts spring-break photos
  could her mother see update #2 but not update #1?
```

```
  ACL and photo list must be in the same record
    since PNUTS guarantees order only for updates to same record
  How could a storage system get this wrong?
    No ordering through single master (e.g. Dynamo)

What if Alice's mother does two reads:
  1. reads the ACL
  2. reads the photo list
  #1 could yield old ACL (with mother in it); #2 could yield new photo
  so: mother must read Alice's record just once

how to change record's master if no failures?
  e.g. I move from Boston to LA
  perhaps just update the record, via old master?
    since ID of master region is stored in the record
  old master announces change over MB
  a few subsequent updates might go to the old master SU
    it will reject them, app retries and finds new master?

what about tolerating failures?

app server crashes midway through a set of updates
  not a transaction, so only some of writes will happen
  but master SU/MB either did or didn't get each write
    so each write happens at all regions, or none

SU down briefly, or network temporarily broken/lossy
  (I'm guessing here, could be wrong)
  MB keeps trying until SU acks
    SU shouldn't ACK until safely on disk

SU loses disk contents, or doesn't automatically reboot
  need to restore disk content from SUs at other regions
    1. subscribe to MB feed, and save them for now
    2. copy content from SU at another region
    3. replay saved MB updates
  puzzle:
    how to ensure we didn't miss any MB updates for this SU?
      e.g. subscribe to MB at time=100, but source SU only saw through 90?
    will replay apply updates twice? is that harmful?
    paper mentions sending checkpoint message through MB
      maybe fetch copy as of when the checkpoint arrived
      and only replay after the checkpoint
      BUT no ordering among MB streams from multiple regions

MB crashes after accepting update
  logs to disks on two MB servers before ACKing
  MB recovery looks at log, (re)sends logged msgs
  record master SU maybe re-sends an update if MB crash before ACK
    maybe record version #s will allow SUs to ignore duplicate

record's master region loses network connection
  can other regions designate a replacement RM?
    no: original RM's MB may have logged updates, only some sent out
  do other regions have to wait indefinitely? yes
    this is one price of ordered updates / strict-ish consistency

now performance

evaluation focuses on latency, not throughput
  why are they apparently worried about latency but not throughput?
  maybe throughput can be indefinitely increased by adding more SUs
  whereas latency is harder to reduce

big performance question: why isn't the MB a terrible bottleneck?
  all writes funnel through a single pair of MBs
  which log every write to disk
  i do not know how they avoid this MB bottleneck
```



```
5.2: time for an insert while busy
  probably measuring time until client sees commit reply from MB
  depends on how far away Record Master is
  RM local: 75.6 ms
  RM nearby: 131.5 ms
  RM other coast: 315.5 ms

Why 75 ms for write to local RM?
  speed-of-light delay?
    no: local
  queuing, waiting for other client's operations?
    no: they imply 100 clients was max that didn't cause delay to rise
  End of 5.2 suggests 40 ms of 75 ms in in SU
    how could it take 40 ms?
      each key/value is one file?
      creating a file takes 3 disk writes (directory, inode, content)?
    what's the other 35 ms?
      MB disk write?

5.3 / Figure 3: effect of increasing request rate
  what do we expect for graph w/ x-axis req rate, y-axis latency?
    system has some inherent capacity, e.g. total disk seeks/second
    for lower rates, constant latency
    for higher rates, queue grows rapidly, avg latency blows up
  blow-up should be near max capacity of h/w
    e.g. # disk arms / seek time
  we don't see a blow-up in Figure 3
    end of 5.3 says clients too slow
  text says max possible rate was about 3000/second
    10% writes, so 300 writes/second
    5 SU per region, so 60 writes/SU/second
    about right if each write does a random disk I/O
    but you'll need lots of SUs for millions of active users
  mystery: how is MB able to log 300 writes/second?
    maybe each region's MB only logs 1/3 of writes, i.e. 100 writes/second?

stepping back, what were PNUTS key design decisions?
  1. replication of all data at multiple regions
     fast reads, slow but async writes
  2. relaxed consistency -- stale reads
     needed if you want fast reads and async writes
  3. sequence all writes thru record master
     good for consistency; bad for latency
  4. only single-row write semantics (order and t-a-s-w mini-transactions)
     simplifies system; maybe awkward for applications

Next: Dynamo, a very different design
  no master -- any region can update any value at any time
  eventual consistency
  tree of versions if network partitions
  readers must reconcile versions
```