```
6.824 2020 Lecture 10: Database logging, quorums, Amazon Aurora

Amazon Aurora, Verbitski et al., SIGMOD 2017.

Why are we reading the Aurora paper?
  successful recent cloud service, solves serious problems for customers
  big payoff for good design (Table 1)
  shows limits of general-purpose storage abstraction
  many tidbits about what's important in real-world cloud infrastructure

Here's my understanding of the story that led to Aurora.

Amazon EC2 -- cloud computing, aimed at web sites
  [VMM, linux guests, www servers &c]
  [web clients, EC2 www servers, EC2 DB servers w/ local disk]
  you can rent virtual machines -- "instances"
  run in Amazon's datacenters, on their physical machines
  virtual local disk stored on directly attached physical disk
  customers run their www servers and DBs on EC2
  EC2 is great for www servers
    more load -> rent more EC2 instances
    crash -> who cares, stateless, data in DB
  EC2 is not ideal for DB e.g. MySQL
    limited scaling options -- just read/only DB replicas
    limited fault-tolerance; if phys machine dies, disk dies with it
      periodic DB backups to Amazon's "S3" bulk storage service

Amazon EBS (Elastic Block Store)
  [diagram of EC2 instance, two EBS servers -- left half of Figure 2]
  block storage that's still available even if an EC2 instance crashes
  looks like a disk to EC2 instances, mount e.g. Linux ext4 on EBS volume
  implemented as pairs of servers with disk drives, chain replication,
    paxos-based configuration manager
  both replicas are in the same "availability zone" (AZ)
    AZ = machine room or datacenter
    for speed, since all client writes have to wait for both EBS servers
  for a DB on EC2, much better fault-tolerance than locally-attached storage!
    if DB EC2 instance crashes, just re-start it on another EC2 instance
    and attach to the same EBS volume
  note: only one EC2 instance can use a given EBS volume at a time
    EBS is not shared storage

DB-on-EBS shortcomings
  lots of data sent over network -- both log and dirty data pages
    the data pages are big even if only a few bytes are changed
  not fault-tolerant enough
    two replicas not enough, but a chain with more would be too slow!
    both replicas are in same AZ
      vulnerable to fire, flood, power failure, broken Internet connection

Paper assumes you know how DB works, how it uses storage.
  here's an overview of a generic transactional DB
  single machine
  example bank transaction -- transfer $10 from x's account to y's account
    x = x + 10
    y = y - 10
  locks x and y, released after commit finishes
  data in b-tree on disk
  cached data pages
  DB server modifies only cached data pages as transaction runs
  and appends update info to WAL (Write-Ahead Log)
    the paper's "redo log" == MySQL's INNODB transaction log.
  log entries:
    LSID  TID  key  old  new
    101     7    x  500  510     (generated by x = x + 10)
    102     7    y  750  740     (generated by y = y - 10)
    103     7  commit            (transaction finished)
  commit once WAL safe on disk
    release locks on x and y
    reply to client
  later write modified data pages from cache to disk
    to free log space, allow cache eviction
    crucial to performance to delay -- write absorbtion, since pages are big
  crash recovery
    replay "new" values for all committed xactions in log (redo)
    replay "old" values for uncommitted xactions in log (undo)

Next idea: Figure 2
  This is Amazon's Multi-AZ RDS
    database-as-a-service, rather than customers running their own on EC2
  Goal: better fault tolerance by cross-AZ replication
  EBS-level mirroring of DB's EBS volume
    Every database server EBS write sent to local EBS *and*
      mirror EBS driven by another EC2 instance.
    Mirroring log and all dirty data pages.
  DB write has to wait for all four EBS replicas to respond.
    Long delays, lots of data.
    But more fault-tolerant due to 2nd AZ.

Aurora big picture
  [diagram]
  DB clients (e.g. web servers).
  One database server (in an EC2 instance).
    Dedicated to one customer.
  Six storage servers store six replicas of a volume.
    Each volume is private to the one database server.

Aurora uses two big ideas:
  Quorum writes for better fault-tolerance without too much waiting
  Storage servers understand how to apply DB's log to data pages
    So only need to send (small) log entries, not (big) dirty pages
    Sending to many replicas, but not much data

Table 1 summarizes results -- 35x throughput increase!
  Hard to guess is hardware resources are equivalent.
  Hard to tell how much of win from quorum writes vs sending log only.
  Maybe mostly due to much less data being sent.
  Apparently network capacity is a primary limiting factor.
    Since the design reduces network use, but increases CPU and storage use.

What's Aurora's storage fault tolerance goal?
  be able to write even if one AZ entirely dead
  be able to read even if one dead AZ and one more storage server dead
    no writes in this case
    but can still serve r/o requests, and create more replicas
  called "AZ+1" -- read even if failure of one AZ plus one more server
  tolerate temporarily slow replicas smoothly (a big deal)
  repair dead replica very quickly (in case another failure!)

Failure of the database server is a separate issue.
  A new database server is started on a new EC2 instance.
    Who/what decides when to do this?
    How is split brain avoided for the DB server?
    Paper doesn't say much about this.
  New DB server recovers from log + data on the storage servers.

Quorum read/write technique
  this is an old and much-used idea (Gifford SOSP 1979)
  the goal: fault-tolerant storage,
    read latest data even if some failures
  usually applied to simple read/write (put/get) storage
```

```
    N replicas
    writes to any W replicas
    reads from any R replicas
    R + W = N + 1
    [diagram with overlap]
    simplest case: N=3, W=2, R=2
    overlap ensures read quorum includes >= 1 server from previous write quorum
    how to decide which is the most recent copy?
      cannot vote on content!
        only one server in read quorum might be up to date
      writer assigns increasing version numbers to its writes
      storage servers must remember version of each item
      reader takes max version # of the R copies it receives
    what if a read or write can't assemble a quorum?
      keep trying.

  What is the benefit of quorum read/write storage systems?
    In contrast to e.g. chain replication.
    Smooth handling of dead or slow or partitioned storage servers
      No need to wait, no need to detect failure, no timeout
      Important for Amazon for remote AZs and temporarily slow servers
      No risk of split brain among the storage servers
    Can adjust R and W to make reads or writes faster (but not both).
    But:
      Servers not in write quorum must catch up (e.g. Aurora's gossip).
      Can only tolerate min(N-W, N-R) failures.
      Requirement for version numbers makes it most suitable for single writer.

  Note
    Raft uses quorums too
      Leader can't commit unless majority of peers receive
      R/W overlap guarantees new Raft leader sees every committed write
    Raft is more capable:
      Raft handles complex operations, b/c it orders operations.
      Raft changes leaders automatically w/o split brain.

  What N,W,R does Aurora need?
    How about N=3, W=2, R=2?
      With each replica in a different AZ?
      Doesn't fulfil AZ+1 goal, since that leaves only one replica.
    Aurora uses N=6, W=4, R=3
      two replicas in each of three AZs
      one AZ failure leaves a write quorum (4) intact
      one extra failure leaves a read quorum (3) intact

  What does an Aurora quorum write look like?
    Not a classic quorum write:
      DB server's writes to storage servers do *not* modify existing data items!
    A write consists of a new log entry.
      For an in-progress transaction.
      Or a commit log record.
    Aurora sends each log record to all six storage servers.
    Transaction commit waits for all of xaction's log records to be on a quorum.
      Stated as VDL >= transaction's last record.
      So every xaction through this one can be recovered after a crash.
    Commit ==> release locks, reply to client.

  What do storage servers do with incoming log entries ("writes")?
    Just store them, initially.
      As a list of relevant pending log entries per data page.
    So storage server state is
      An [old] version of each data page.
      The list of log records required to bring the page up to date.
      [diagram, data pages for x and y, log entries]
    Storage server applies log records in the background.
      Or when needed by a read.
```

```
What does an ordinary Aurora read look like?
  Not a quorum read!
  The Aurora DB server needs a data page, due to a cache miss.
    Writes are log entries; reads yield data pages.
  Needs to find a storage server that has all the relevant log entries.
  Database server tracks each storage server's SCL (Segment Complete LSN).
    Each storage server has all log entries <= its SCL.
    Reports back to the database server.
  DB server reads from any storage server whose SCL >= highest committed LSN.
    That storage server may need to apply some log records to the data page.

When does Aurora use read quorums?
  Only during crash recovery.
    Crash means database server crashed, re-started on a different EC2 instance.
    New database server uses existing six storage servers.
  What has to be done for recovery?
    Nothing much for committed transactions -- already on a write quorum.
    What about transactions interrupted by a crash?
      Cannot be finished b/c database server has lost its execution state.
  DB finds highest log record such that log is complete up to that point.
    Log record exists if it's on any reachable storage server.
    The VCL (Volume Complete LSN) of Section 4.1.
  DB tells storage servers to delete all log entries after the VCL.
  DB issues compensating "undo" log entries for transactions that
    started before VCL but didn't commit.

What if the database is too big for a replica to fit into one storage server?
  Data pages are sharded into 10-GB Protection Groups (PGs).
  Each PG is separately replicated as six "segments" (on six replicas).
    Different PGs likely to be on different sets of six storage servers.
  [DB server, two PGs on 12 storage servers]

How is the *log* stored w.r.t. PGs?
  DB server sends each PG just the log records relevant to that PG's data pages.

How to restore replication if a storage server permanently dies?
  Needs to happen quickly to minimize exposure to another failure.
  The dead storage server held replicas of perhaps 1000 PGs (10 TB).
    Copying 10 TB from one server to another would take 10,000 seconds!
  For the various PGs,
    their *other* replicas are likely on different storage servers, and
    their newly created replicas can also be on different storage server.
    [S stores PGs a, b, and c, each w/ replicas on different other servers]
  So each PG can re-replicate fully in parallel:
    Each PG can be sent from a different source server, to a different dst.
    Up to the number of storage servers.
  If lots of storage servers, the *whole* job takes only 10 seconds.

What about the read-only replicas ("Replica Instance") in Figure 3?
  Clients can send read-only requests to them.
    To reduce load on main database server, which is likely a bottleneck.
    Results may not reflect recent committed updates!
  Read-only replicas read data pages from the storage servers (but don't write).
    And cache these data pages.
  Main DB server sends log to replicas.
    Replicas use log to keep cached pages up to date.
  Problem: read-only replicas do not know what is locked by transactions.
    Solution: replicas ignore or un-do writes of uncommitted transactions.
      They can tell from commit indications in the log stream.
  Problem: need to prevent replica from seeing B-Tree &c in intermediate state
    Solution: atomic mini-transactions within larger transactions.

What are the take-away lessons?
  How write-ahead logging interacts with storage systems.
  Quorums for smooth handling of slow/failed replicas.
  Cross-layer optimization -- DB and storage servers.
    Yields great performance improvements.
    But storage isn't general-purpose.
  Tidbits about primary concerns for cloud infrastructure.
    AZ failure, thus need multi-AZ replication.
    Transient slowness of individual replicas.
    Network a worse constraint than CPU or storage.
```