Cloud Databases – Amazon Aurora

Prof. Reza Farivar

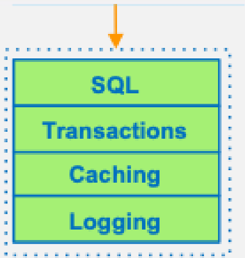
CLOUD COMPUTING APPLICATIONS

Relational Database on the Cloud

- Traditional RDBMS rely on B+Trees, replication, etc. to optimize usage on one or a few servers
- Cloud brings many new things to the table
  - Backend storage
  - Network
  - Worldwide Scalability
- How can we optimize RDBMS for the cloud?
- First step: separate storage layer from the transactional logic
  - Decoupling storage from compute
- Deuteronomy
  - Transaction Component (TC) provides concurrency control and recovery
  - Data Component (DC) provides access methods on top of LLAMA, a latch-free log-structured cache and storage manager.
- Aurora, CosmosDB both inspired by Deuteronomy

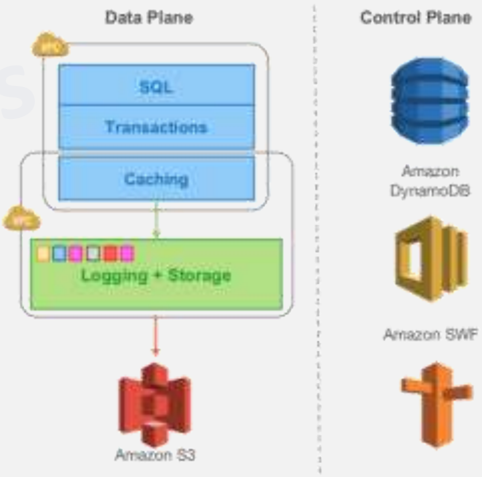
Amazon AWS Aurora

- Optimized DB engine, built from MySQL (and later PostgreSQL), with a distributed storage layer
- API compatible with MySQL or Postgres
  - i.e. you can use an existing application
- Separate storage and compute
  - Query processing, transactions, concurrency, buffer cache, and access
  - Logging, storage, and recovery that are implemented as a scale out service.
- Move caching and logging layers into a purpose-built, scale-out, self-healing, multitenant, database-optimized storage service



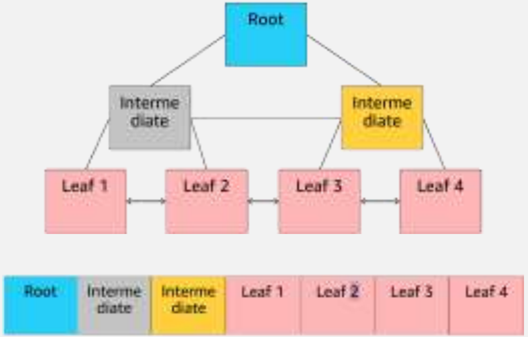
Amazon AWS Aurora

- Each instance still includes most of the components of a traditional kernel (query processor, transactions, locking, buffer cache, access methods and undo management)
- Several functions (redo logging, durable storage, crash recovery, and backup/restore) are off-loaded to the storage service

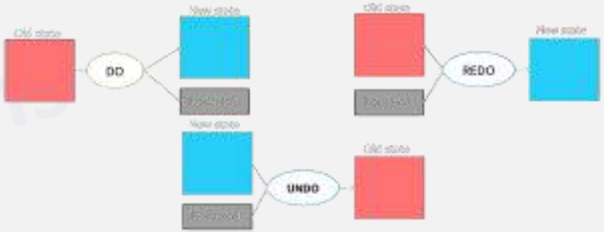


Redo Logging

- Traditional relational databases organize data in *pages* (e.g. 16KB), and as pages are modified, they must be periodically flushed to disk
  - B+ Tree
- For resilience against failures and maintenance of ACID semantics, page modifications are also recorded in *do-redo-undo log records*, which are written to disk in a continuous stream.
- rife with inefficiencies.
  - E.g. a single logical database write turns into multiple (up to five) physical disk writes, resulting in performance problems.
  - Write Amplification
  - combat the write amplification problem by reducing the frequency of page flushes
  - This in turn worsens the problem of crash recovery duration



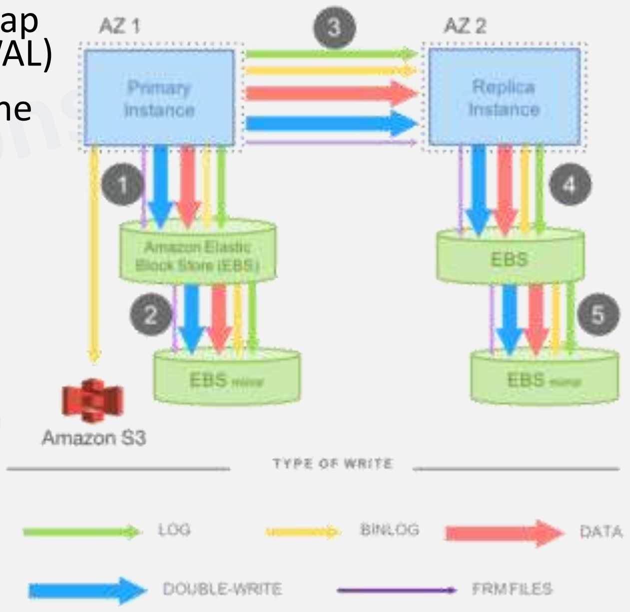
Redo Logging



- Traditional relational databases organize data in *pages* (e.g. 16KB), and as pages are modified, they must be periodically flushed to disk
  - B+ Tree
- For resilience against failures and maintenance of ACID semantics, page modifications are also recorded in *do-redo-undo log records*, which are written to disk in a continuous stream
  - redo log record : difference between the after and the before-image of a page
- rife with inefficiencies.
  - E.g. a single logical database write turns into multiple (up to five) physical disk writes, resulting in performance problems.
    - Write Amplification
  - Combat the write amplification problem by reducing the frequency of page flushes
    - This in turn worsens the problem of crash recovery duration

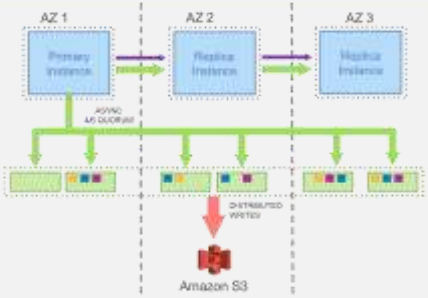
Burden of Amplified Writes

- A system like MySQL writes data pages to objects it exposes (e.g., heap files, b-trees etc.) as well as redo log records to a write-ahead log (WAL)
- The writes made to the primary EBS volume are synchronized with the standby EBS volume using software mirroring
- Data needed to be written in step 1:
  - redo log (typically a few bytes, transaction commit requires the log to be written)
  - binary (statement) log that is archived to Amazon S3 to support point-in-time restores
  - modified data pages (the data page write may be deferred, e.g. 16KB)
  - a second temporary write of the data page (double-write) to prevent torn pages (e.g. 16KB)
  - metadata (FRM) files
- Steps 1, 3, and 5 are sequential and synchronous
  - Latency is additive because many writes are sequential
  - 4/4 write quorum requirement and is vulnerable to failures and outlier performance
- Different writes representing the same information in multiple ways



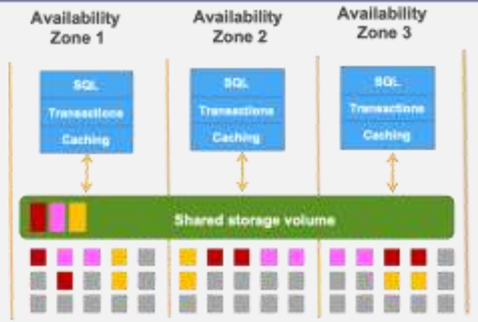
Log is the database

- In Amazon Aurora, the log is the database
- Database instances write redo log records to the distributed storage layer, and the storage takes care of constructing page images from log records on demands from the database
  - Write performance is improved due to the elimination of write amplification and the use of a scale-out storage fleet
  - 5x write IOPS on the SysBench benchmark compared to Amazon RDS for MySQL running on similar hardware
  - Database crash recovery time is cut down, since a database instance no longer has to perform a redo log stream replay
    - From 27 seconds down to 7 seconds according to one benchmark

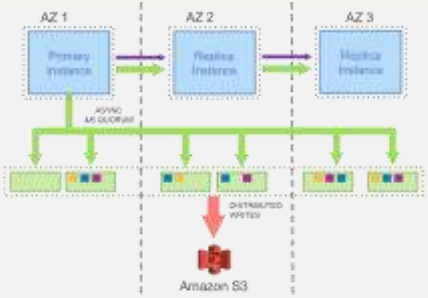


Aurora Replication and Quorum

- Everything fails all the time
  - The traditional approaches of blocking I/O processing until a failover can be carried out—and operating in "degraded mode" until recovery —are problematic at scale
    - in a large system, the probability of operating in degraded mode approaches 1
- Aurora uses quorums to combat the problems of component failures and performance degradation
  - Write to as many replicas as appropriate to ensure that a quorum read always finds the latest data
- Goal is Availability Zone+1: tolerate a loss of a zone plus one more failure without any data durability loss, and with a minimal impact on data availability
  - 4/6 quorum
  - For each logical log write, issue six physical replica writes
    - Write operation successful when four of those writes complete
    - Instances only write redo log records to storage
    - Typically 10s to 100s of bytes, makes a 4/6 write quorum possible without overloading the network
- If a zone goes down and an additional failure occurs, can still achieve read quorum (3/6), and then quickly regain the ability to write by doing a *fast repair*



Offloading Redo Processing to Storage



- In Aurora, the only writes that cross the network are redo log records
  - No pages are ever written from the database tier, not for background writes, not for checkpointing, and not for cache eviction
- log applicator is pushed to the storage tier to generate database pages in background or on demand
  - Generating each page from the complete chain of its modifications from the beginning of time is prohibitively expensive
  - Continually materialize database pages in the background to avoid regenerating them from scratch on demand every time
- The storage service can scale out I/Os in an embarrassingly parallel fashion without impacting write throughput of the database engine
- primary only writes log records to the storage service and streams those log records as well as metadata updates to the replica instances
- database engine waits for acknowledgements from 4 out of 6 replicas in order to satisfy the write quorum

Aurora Fast Repair

- Amazon Aurora approach to replication: based on sharding and scale-out architecture
- An Aurora database volume is logically divided into 10-GiB logical units (*protection groups*), and each protection group is replicated six ways into physical units (*segments*)
- When a failure takes out a segment, the repair of a single protection group only requires moving ~10 GiB of data, which is done in seconds.
- When multiple protection groups must be repaired, the entire storage fleet participates in the repair process.
  - Massive bandwidth to complete the entire batch of repairs
- A zone loss followed by another component failure → Aurora may lose write quorum for a few seconds for a given protection group
  - Recovery is quick

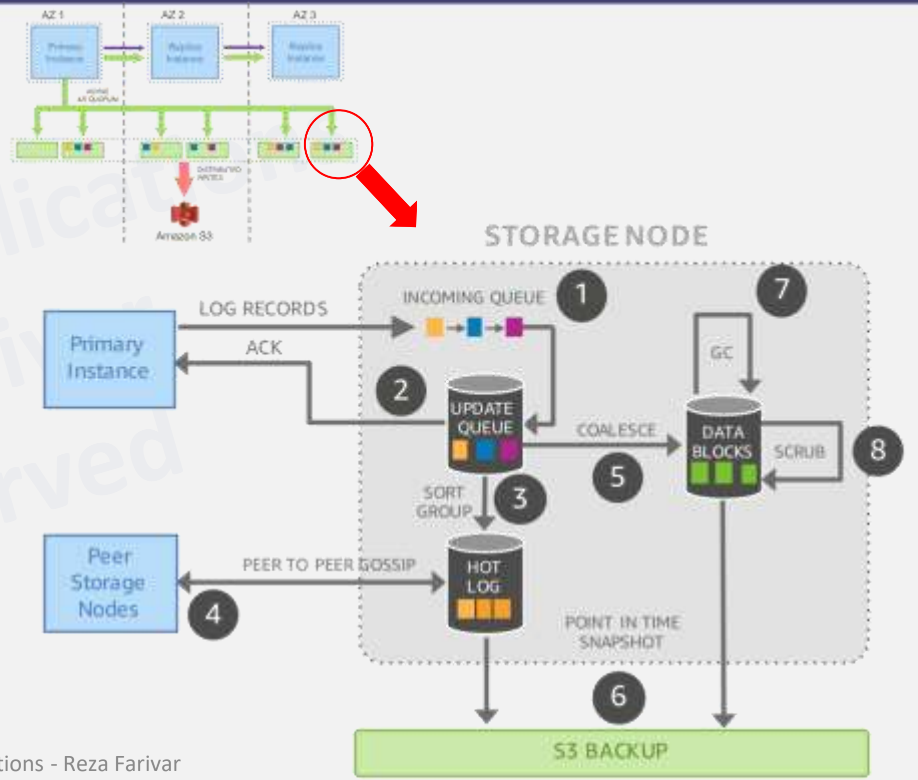
Quorum Reads

- A quorum read is expensive, and is best avoided
- We do not need to perform a quorum read on routine page reads
  - It always knows where to obtain an up-to-date copy of a page
  - The client-side Aurora storage driver tracks which writes were successful for which segments
  - The driver tracks read latencies, and always tries to read from the storage node that has demonstrated the lowest latency in the past
- The only scenario when a quorum read is needed is during recovery on a database instance restart

Amazon Aurora Storage Nodes

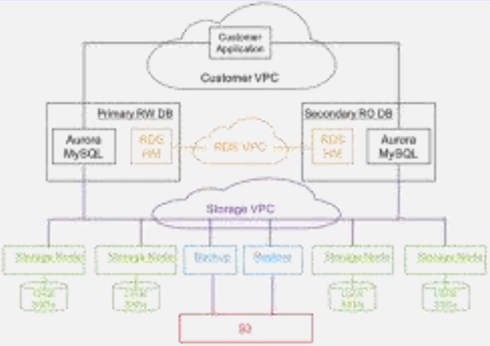
1. Receive log records and add to in-memory queue
2. persist record on disk and acknowledge, ACK to the database
3. Organize records and identify gaps in log since some batches may be lost
4. Gossip with peers to fill in holes
5. Coalesce log records into new page versions
6. Periodically stage log and new page versions to S3
7. Periodically garbage-collect old versions
8. Periodically validate CRC codes on blocks

- Each of the steps above are asynchronous
- Only steps (1) and (2) are in the foreground path potentially impacting latency



Database Engine Implementation

- The database engine is a fork of “community” MySQL/InnoDB and diverges primarily in how InnoDB reads and writes data to disk
  - In community InnoDB, a write operation results in data being modified in buffer pages, and the associated redo log records written to buffers of the WAL in LSN order
  - On transaction commit, the WAL protocol requires only that the redo log records of the transaction are durably written to disk
  - The actual modified buffer pages are also written to disk eventually through a double-write technique to avoid partial page writes
  - These page writes take place in the background, or during eviction from the cache, or while taking a checkpoint
- In addition to the IO Subsystem, InnoDB also includes the transaction subsystem, the lock manager, a B+-Tree implementation and the associated notion of a “mini transaction” (MTR).
  - An MTR is a construct only used inside InnoDB and models groups of operations that must be executed atomically (e.g., split/merge of B+-Tree pages).
- Concurrency control is implemented entirely in the database engine without impacting the storage service

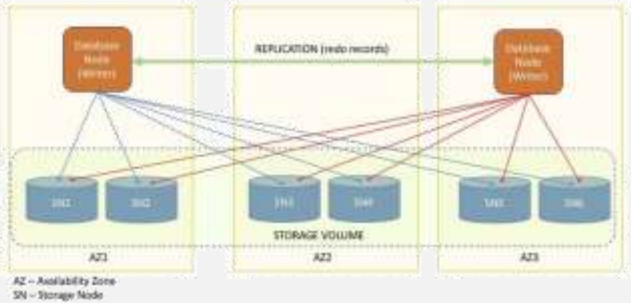


Aurora and Consensus

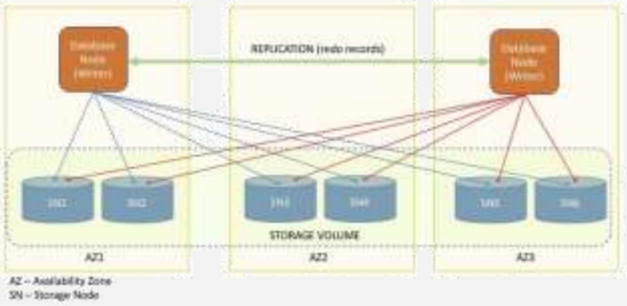
- Aurora leverages only quorum I/Os, locally observable state, and monotonically increasing log ordering to provide high performance, non-blocking, fault-tolerant I/O, commits, and membership changes
- Aurora is able to avoid much of the work of consensus by recognizing that, during normal forward processing of a system, there are local oases of consistency
- Using backward chaining of redo records, a storage node can tell if it is missing data and gossip with its peers to fill in gaps
- Using the advancement of segment chains, a database instance can determine whether it can advance durable points and reply to clients requesting commits
- The use of monotonically increasing consistency points – SCLs, PGCLs, PGMRPLs, VCLs, and VDLs – ensures the representation of consistency points is compact and comparable
    - These may seem like complex concepts but are just the extension of familiar database notions of LSNs and SCNs.
- The key invariant is that the log only ever marches forward.

Aurora Multi-Master

- For high availability and ACID transactions across a cluster of database nodes with configurable read after write consistency
- With single-master Aurora, a failure of the single writer node requires the promotion of a read replica to be the new writer
- In the case of Aurora Multi-Master, the failure of a writer node merely requires the application using the writer to open connections to another writer
- When designing for high availability, make sure that you are not overloading writers
- Conflicts arise when concurrent transactions or writes executing on different writer nodes attempt to modify the same set of pages



Aurora Multi-Master Replication and Quorom



1. The application layer starts a write transaction
2. For cross-cluster consistency: The writer node proposes the change to all six storage nodes
3. Each storage node checks if the proposed change conflicts with a change in flight or a previously committed change and either confirms the change or rejects it
   - Each storage node compares the LSN (think of this as a page version) of the page submitted by the writer node with the LSN of the page on the node
   - It approves the change if they are the same and rejects the change with a conflict if the storage node contains a more recent version of the page
4. If the writer node that proposed the change receives a positive confirmation from a quorum of storage nodes:
   1. First, it commits the change in the storage layer, causing each storage node to commit the change
   2. It then replicates the change records to every other writer node in the cluster using a low latency, peer-to-peer replication protocol
5. The peer writer nodes, upon receiving the change, apply the change to their in-memory cache (buffer pool)
6. If the writer node that proposed the change does not receive a positive confirmation from a quorum of storage nodes, it cancels the entire transaction and raises an error to the application layer (The application can then retry the transaction)
7. Upon successfully committing changes to the storage layer, writer nodes replicate the redo change records to peer writer nodes for buffer pool refresh in the peer node

Aurora vs. RDS

- RDS offers a greater range of database engines and versions than Aurora RDS
- Aurora RDS offers superior performance to RDS due to the unique storage subsystem
- Aurora RDS offers superior scalability to RDS due to the unique storage subsystem
- The pricing models differ slightly between RDS and Aurora RDS, but Aurora RDS is generally a bit more expensive to implement for the same database workload
- Aurora RDS offers superior high availability to RDS due to the unique storage subsystem
- According to AWS, Aurora offers five times the throughput of standard MySQL, performance on-par with commercial databases, but at one-tenth the cost
    - It should be noted that these numbers are AWS marketing claims.
    - House of Brick has found the cost claims to be roughly correct when compared with Oracle Enterprise Edition deployments, but the performance advantage of Aurora in real-world scenarios is closer to 30%

http://houseofbrick.com/aws-rds-mysql-vs-aurora-mysql/