Amazon Aurora: Design Considerations + On Avoiding Distributed Consensus for I/Os, Commits, and Membership Changes

- March 20, 2022

Amazon Aurora is a high-throughput cloud-native relational database. I will summarize its design as covered by the Sigmod 17 and Sigmod 18 papers from the Aurora team. Aurora uses MySQL or PostgreSQL for the database instance at top, and decouples the storage to a multi-tenant scale-out storage service. In this decoupled architecture, each database instance acts as a SQL endpoint and supports query processing, access methods, transactions, locking, buffer caching, and undo management. Some database functions, including redo logging, materialization of data blocks, garbage collection, and backup/restore, are offloaded to the storage nodes.

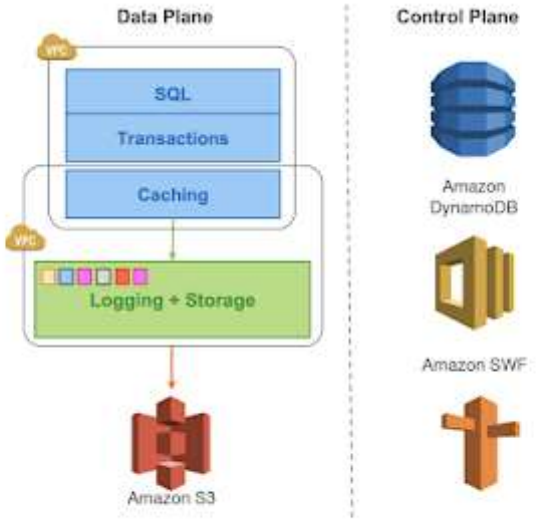


Figure 1: Move logging and storage off the database engine

A big innovation in Aurora is to do the replication among the storage nodes by pushing the redo log; this reduces networking traffic and enables fault-tolerant storage that heals without database involvement.

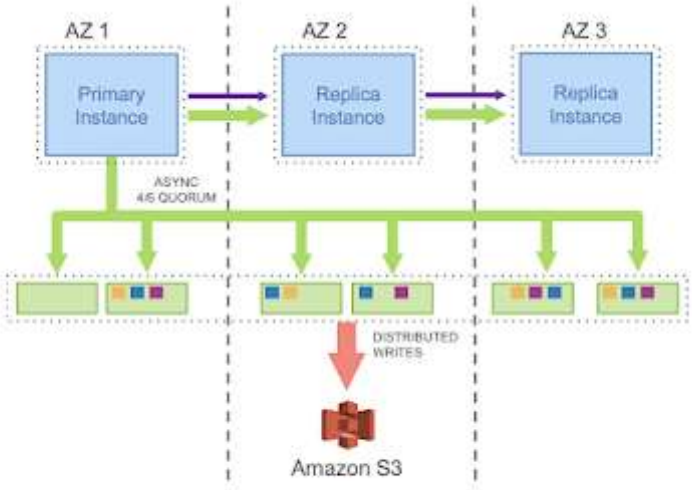


Figure 3: Network IO in Amazon Aurora

In contrast to CockroachDB and FoundationDB, Aurora manages not to use consensus at all. It uses a primary secondary failover at the compute, and decouples the storage and does interesting scaleout protocol at that layer. By leveraging the ample replication at scaleout it also makes the failover at compute fault-tolerant and quite fast. By using quorums, locally observable state, and monotonically increasing log ordering, Aurora avoids distributed consensus for commits, reads, replication, and membership changes. We will discuss this below in the Redo Log and Crash Recovery sections in detail.

Aurora supports "AZ+1" failures (AZ means availability zone). It maintains 6 copies of data, spread across 3 AZs, such that there are 2 replicas per AZ, a 4/6 forms a write quorum, and a 3/6 forms a read/recovery quorum.

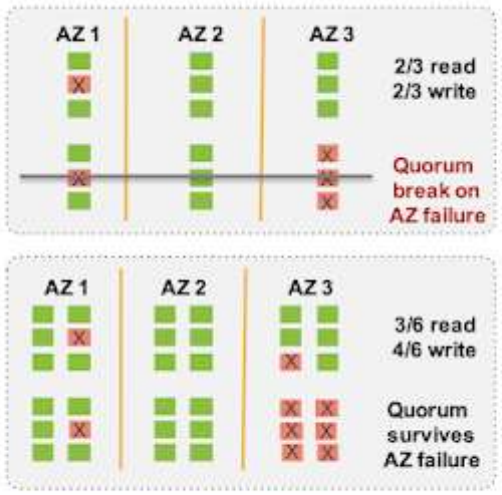


Figure 1: Why are 6 copies necessary ?

Aurora also tackles the double fault problem and tries to keep a write quorum going and not losing availability in the presence of two uncorrelated faults in a small period after an AZ unavailability. It is difficult, past a point, to reduce the probability of MTTF (mean time to failure) on independent failures. They instead focus on reducing MTTR (mean time to repair), to shrink the window of vulnerability to a double fault. They do so by partitioning the database volume into small fixed size segments, 10GB in size. It is faster to recover small size segments than big size segments. Each segment is called a **protection group** and, as mentioned above, is spread across 3 AZs, with 2 replicas per AZ.

Redo log deep dive

While the redo log is segmented across 10GB and spread across storage nodes, the **Log Sequence Number (LSN)** space is common across the database volume. The LSN is monotonically increasing, and is allocated by the database instance (i.e., the primary instance).

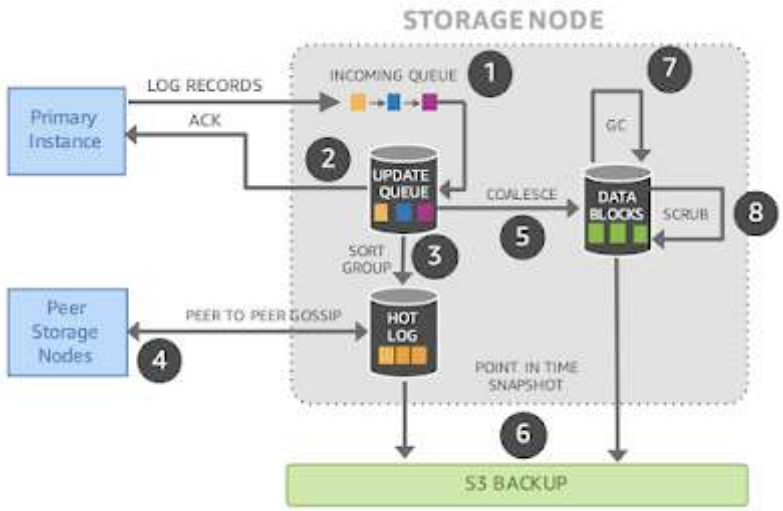


Figure 2: Activity in Aurora Storage Nodes

As a storage node receives new log records, it may locally advance a **Segment Complete LSN (SCL)**, representing the latest point in time for which it knows it has received all log records. SCL is sent by the storage node as part of acknowledging a write. Once the database instance observes SCL advance at four of six members of the protection group, it is able to locally advance the **Protection Group Complete LSN (PGCL)**, representing the point at which the protection group has made all writes durable.

For example, Figure 3 shows a database with two protection groups, PG1 and PG2, consisting of segments A1-F1 and A2-F2 respectively. In the figure, each solid cell represents a log record acknowledged by a segment, with the odd numbered log records going to PG1 and the even numbered log records going to PG2. Here, PG1's PGCL is 103 because 105 has not met quorum, PG2's PGCL is 104 because 106 has not met quorum, and the database's VCL is 104 which is the highest point at which all previous log records have met quorum.

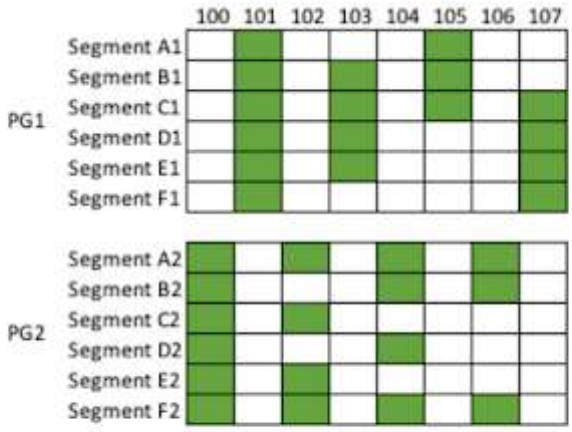


Figure 3: Storage Consistency Points

For a database, it is not enough for individual writes to be made durable, the entire log chain must be complete to ensure recoverability. The database instance also locally advances a **Volume Complete LSN (VCL)** once there are no pending writes preventing PGCL from advancing for one of its protection groups.

No consensus is required to advance SCL, PGCL, or VCL: all that is required is bookkeeping by each individual storage node and local ephemeral state on the database instance based on the communication between the database and storage nodes. Since Aurora uses a single writer (i.e., the primary instance), accept versus commit separation in Paxos is not needed in Aurora. In Aurora, everything received at a replica node is labeled as an immediate commit. If needed, the primary instance fail-overs, and leveraging the quorum, the epoch number is increased and recovery is performed, and the new primary keeps advancing the log. We discuss how that works next.

Crash recovery

The time saved in the normal forward processing of commits using local transient state must be paid back by additional work done for re-establishing consistency

upon crash recovery. This is a trade worth making since commits are many orders of magnitude more common than crashes.

When opening a database volume, either for crash recovery or for a normal startup, the database instance must be able to reach at least a read quorum for each protection group comprising the volume. The database instance can then locally re-compute PGCLs and VCL for the database by finding read quorum consistency points across SCLs.

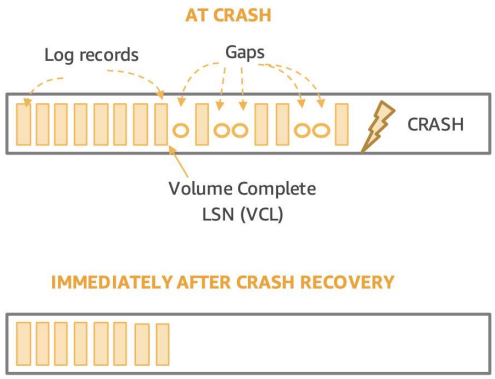


Figure 4: Log truncation during crash recovery

If Aurora is unable to establish write quorum for one of its protection groups, it initiates repair from the available read quorum to rebuild the failed segments. Once the volume is available for reads and writes, Aurora increments an *epoch* in its storage metadata service and records this volume epoch in a write quorum of each protection group comprising the volume. The volume epoch is provided as part of every read or write request to a storage node. Storage nodes will not accept requests at stale volume epochs. This boxes out old instances with previously open connections from accessing the storage volume after crash recovery has occurred. Some systems use leases, but Aurora, rather than waiting for a lease to expire, just changes the locks on the door.

Making reads efficient

Aurora uses read views to support snapshot isolation using Multi-Version Concurrency Control (MVCC). A read view establishes a logical point in time before which a SQL statement must see all changes and after which it may not see any changes other than its own.

Aurora does not do quorum reads. Through its bookkeeping of writes and consistency points, the database instance knows which segments have the last durable version of a data block and can request it directly from any of those segments. Aurora MySQL does this by establishing the most recent SCN and a list

of transactions active as of that LSN. Reads are one of the few operations in Aurora where threads have to wait. Unlike writes (which can stream asynchronously to storage nodes) or commits (where a worker can move on to other work while waiting for storage to acknowledge), a thread needing a block not in cache typically must wait for the read I/O to complete before it can progress.

Internally, within an Aurora cluster, physical replication is used (as it is faster and more efficient than logical replication). Aurora read replicas attach to the same storage volume as the writer instance. They receive a physical redo log stream from the writer instance and use this to update only data blocks present in their local caches. Redo records for uncached blocks can be discarded, as they can be read from the shared storage volume. Decoupling database from the storage allows Aurora to horizontally scale read instances.

Aurora uses three invariants to manage structural consistency of the replicas. First, replica read views must lag durability consistency points at the writer instance. This ensures that the writer and reader need not coordinate cache eviction. Second, structural changes to the database, for example B-Tree splits and merges, must be made visible to the replica atomically. This ensures consistency during block traversals. Third, read views on replicas must be anchorable to equivalent points in time on the writer instance. This ensures that snapshot isolation is preserved across the system.

Failures and quorum membership

Quorum membership is also managed leveraging the quorum itself using epoch numbers. Membership epochs enable updating membership without complex consensus, fence out others without waiting for lease expiry, and operate using the same failure tolerance as quorum reads and writes themselves. Clients with stale membership epochs have their requests rejected and must update membership

information.

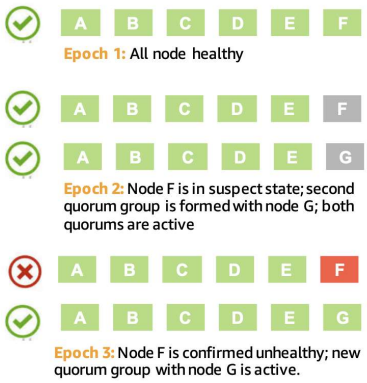


Figure 5: Quorum Membership Changes

Figure 5 illustrates how to replace segment F with segment G. Rather than attempting to directly transition from ABCDEF to ABCDEG, the transition is made in two steps. First, G is added to the quorum, moving the write set to 4/6 of ABCDEF AND 4/6 of ABCDEG. The read set is therefore 3/6 of ABCDEF OR 3/6 of ABCDEG. If F comes back, it is possible to make a second membership change back to ABCDEF. That quorum subset met the write quorum and is an available next step. If F continues to be down once G has completed hydrating from its peers, it is possible to make a membership change to ABCDEG.

Let us now consider what happens if E also fails while we are replacing F with G, and we wish to replace it with H. In this case, we would move from a write quorum set of ((4/6 of ABCDEF AND 4/6 of ABCDEG) AND (4/6 of ABCDFH AND 4/6 of ABCDGH)). As with a single failure, I/Os can proceed, the operation is reversible, and the membership change can occur with an epoch increment.

This simple quorum membership scheme is used for handling unexpected failures, heat management, as well as planned software upgrades.

Using Quorum Sets to Reduce Costs

In Aurora, a protection group is composed of three full segments, which store both

redo log records and materialized data blocks, and three tail segments, which contain redo log records alone. Since most databases use much more space for data blocks than for redo logs, this yields a cost amplification closer to 3 copies of the data rather than a full 6 while satisfying the requirement to support AZ+1 failures.

The use of full and tail segments changes the construction of read and write sets. The write quorum is 4/6 of any segment OR 3/3 of full segments. The read quorum is therefore 3/6 of any segment AND 1/3 of full segments. In practice, this means that Aurora writes log records to the same 4/6 quorum. At least one of these log records arrives at a full segment and generates a data block. Data is read from a full segment, using the optimization described earlier to avoid quorum reads.

cloud computing databases fault-tolerance paper-review

Kyle Kingsbury said...

> Aurora avoids distributed consensus for commits, reads, replication, and membership changes.

So this is something I've always wondered about in Aurora, and never fully understood: if they're not doing consensus for commits or reads etc... what happens when consensus *isn't* satisfied? Surely one of the three consensus properties must (sometimes) fail to hold: safety, liveness, and nontriviality. It sounds like liveness and nontriviality *are* preserved, so... does that leave Aurora violating consensus safety? If so, then somewhere there's got to be learners--perhaps clients, perhaps storage nodes, depending on where you want to draw the boundaries--which *disagree* on the contents or order of transactions. Does this manifest as user-observable consistency violations? If not, why?

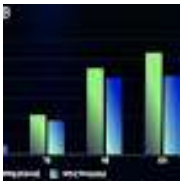
March 21, 2022 at 10:18 AM

Post a Comment

Popular posts from this blog

Graviton2 and Graviton3

- *December 04, 2021*



What do modern cloud workloads look like? And what does that have to do with new chip designs? I found these gems in Peter DeSantis's ReInvent20 and ReInvent21 talks. These talk ...

READ MORE

Foundational distributed systems papers

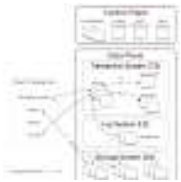
- *February 27, 2021*

I talked about the importance of reading foundational papers last week. To followup, here is my compilation of foundational papers in the distributed systems area. (I focused on the core distributed systems area, and did not cover ...

READ MORE

FoundationDB: A Distributed Unbundled Transactional Key Value Store (Sigmod 2021)

- *March 10, 2022*



This paper from Sigmod 2021 presents FoundationDB, a transactional key-value store that supports multi-key strictly serializable transactions across its entire key-space. Fo ...

READ MORE

Your attitude determines your success

- *March 13, 2021*

This may sound like a cliche your dad used to tell, but after many years of going through new areas, ventures, and careers, I find this to be the most underrated career advice. This is the number one advice I would like my kids to inte ...

READ MORE

Learning a technical subject

- *December 18, 2021*

I love learning. I wanted to write about how I learn, so I can analyze if there is a method to this madness. I will first talk about what my learning process looks like in abstract terms, and then I'll give an analogy to make things more con ...

READ MORE

Progress beats perfect

- *August 06, 2021*

This is a favorite saying of mine. I use it to motivate myself when I feel disheartened about how much I have to learn and improve. If I do a little every day or every week, I will get there. If I ge ...

READ MORE

Cores that don't count

- *June 06, 2021*

This paper is from Google and appeared at HotOS 2021 . There is also a very nice 10 minute video presentation for it. So Google found fail-silent Corruption Execution Errors (CEEs) at CPU/cores. This is interesting because we tl ...

READ MORE

Learning about distributed systems: where to start?

- *June 10, 2020*

This is definitely not a "learn distributed systems in 21 days" post. I recommend a principled, from the foundations-up, studying of distributed systems, which will take a good three months in the first pass, and many more months to build (...

READ MORE

CockroachDB: The Resilient Geo-Distributed SQL Database

- March 04, 2022

This paper appeared in Sigmod 2020. Here is a link to the 10 minute, but extremely useful, Sigmod presentation . There is also a 1 hour extended presentation . CockroachDB is open s(...

READ MORE

Silent data corruptions at scale

- June 12, 2021

This paper from Facebook (Arxiv Feb 2021) is referred in the Google fail-silent Corruption Execution Errors (CEEs) paper as the most related work. Both papers discuss the same phen ...

READ MORE

Murat Demirbas

MURAT

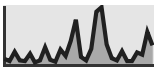
I am a principal applied scientist at AWS S3 Automated Reasoning Group. On leave as a computer science and engineering professor at SUNY Buffalo. I work on distributed systems, distributed consensus, and cloud computing. You can follow me on Twitter.

VISIT PROFILE

Recent Posts

Topics

Pageviews

 **2913056**