Amazon Aurora Under the Hood: Quorum Membership

by Anurag Gupta | on 23 AUG 2017 | in Amazon Aurora, Aurora, Database | Permalink | Comments | Share

Anurag Gupta runs a number of AWS database services, including Amazon Aurora, which he helped design. In this under the hood series, Anurag discusses the design considerations and technology underpinning Aurora.

This post is the last in a four-part series discussing how Amazon Aurora uses quorums. In the first post, I described the benefits of quorums and the minimum number of members that are needed in the face of correlated failures. In the second post, I discussed how to use logging, cached state, and non-destructive writes to avoid network amplification for reads and writes. In the third post, I talked about how to use more advanced quorum models to reduce the costs of replication. In this last post about quorums, I describe how Amazon Aurora avoids problems when managing quorum membership changes.

Techniques for managing quorum membership changes

Machines fail. When one of the members of a quorum fails, we need to repair the quorum by replacing the node. This can be a complex decision. The other members of a quorum can't tell if the impaired member is encountering a latency blip, experiencing a short-term availability loss for a restart, or is down forever. Network partitions can cause multiple groups of members to try simultaneously to fence each other off.

If you're managing large amounts of persistent state per node, the re-replication of state to repair a quorum can take a long time. In such cases, you might want to be conservative about initiating a repair in case the impaired member is able to return. You can instead optimize for repair time by segmenting state across many nodes. But this then increases the likelihood that you will see failures.

In Aurora, we segment a database volume into 10 GB chunks, using six copies spread across three Availability Zones (AZs). For a current maximum database

Create a Free AWS Account

Search

Search

Posts by Product

Amazon Aurora

AWS Database Migration Service (DMS)

Amazon DynamoDB

Amazon EC2

Amazon ElastiCache

Amazon Elasticsearch Service

AWS IOT

Amazon Kinesis

AWS Lambda

Amazon RDS for MySQL

Amazon RDS for Oracle

by using a lease for a period of time and a consensus protocol such as Paxos to ensure membership at each lease. But Paxos is a heavyweight protocol, and optimized versions result in stalls on large numbers of failures.

Using quorum sets to handle failures
Aurora instead uses quorum sets and database techniques such as logging, rollback, and commit to manage membership changes. Let's consider an individual protection group with the six segments A, B, C, D, E, and F. In this case, the write quorum is any four members out of this set of six, and the read quorum is any three members. As I discussed in my last post, Aurora quorums are more complex than this, but let's keep it simple for now.

Each read and write in Aurora uses a membership epoch, a value that monotonically increases with each membership change. Reads and writes that are at an epoch that is older than the current membership epoch are rejected. In such cases, the caller needs to refresh its understanding of the quorum membership. This is conceptually similar to the notion of log sequence numbers (LSNs) in a redo log. The epoch number and associated change record provide an ordered sequence of changes to membership. Changes to the membership epoch require meeting write quorum just as data writes do. Reads of current membership require meeting read quorum just as data reads do.

Let's continue with our protection group of ABCDEF. Imagine that we think segment F might have failed, and we need to introduce the new segment G. We don't want to fence off F—it may be encountering a temporary failure and might come back quickly. Or it may be processing requests, but for some reason, it's not observable to us. We also don't want to wait to see whether F comes back—that just adds time during which the quorum is impaired and a second fault can occur.

We use quorum sets to solve this. We don't do a membership change directly from ABCDEF to ABCDEG. Instead, we increment the membership epoch and move the quorum set to ABCDEF AND ABCDEG. A write must now successfully acknowledge from four out of the six copies in ABCDEF and also acknowledge from four out of the six copies in ABCDEG. Any four members of ABCDE satisfy both write quorums. The read/repair quorum operates identically, requiring any three acknowledgements from ABCDEF and any three from ABCDEG. Again, any three from ABCDE satisfy both.

When the data has been fully hydrated onto node G, and we decide that we want to fence F off, we again do a membership epoch change and change the quorum set to ABCDEG. The use of an epoch makes this an atomic operation, just as a commit LSN does for redo processing. This epoch change needs to satisfy the current write quorum before it succeeds, requiring acknowledgment from four of six in ABCDEF and four of six in ABCDEG, just like any other update. If node F were to become visible again before G filled itself in, we could also just as easily roll back our change and make a

Amazon RDS for SQL Server

AWS Schema Conversion Tool (SCT)

RSS Feed

Subscribe to this blog's feed

Recent Posts

Introducing Amazon S3 and Microsoft Azure SQL Database Connectors in AWS Database Migration Service

Viewing Amazon Elasticsearch Service Slow Logs

Replicating Amazon EC2 or On-Premises SQL Server to Amazon RDS for SQL Server

Querying on Multiple Attributes in Amazon DynamoDB

Automating Cross-Region and Cross-Account Snapshot Copies with the Snapshot Tool for Amazon Aurora

Automating SQL Caching for Amazon

Note that reads and writes to this quorum happen during a membership change just as they would before or after the change. The change to quorum membership does not block reads or writes. At most, it causes callers with stale membership information to refresh their state and reissue the request to the correct quorum set. And quorum membership changes are non-blocking for both read and write operations.

Of course, any one of ABCDEG might also fail while we're in the process of repairing the quorum by fully hydrating G as a replacement for F. Many membership change protocols do not robustly handle faults during membership change. With quorum sets and epochs, it's easy. Let's consider the case where E also fails and is to be replaced by H. We just need to move to a quorum of ABCDEF AND ABCDEG AND ABCDFH AND ABCDGH. As with the single fault, a write to ABCD satisfies all of these. Membership changes have the same tolerance for failures as reads and writes themselves.

Summary

Using quorum sets for membership changes makes it possible for Aurora to use small segments. This improves durability by reducing Mean Time To Repair (MTTR) and our window of vulnerability to multiple faults. It also reduces costs for our customers. Aurora volumes automatically grow as needed, and small segments allow them to grow in small increments. The use of quorum sets ensures that reads and writes can continue even while membership changes are in-flight.

Making membership decisions reversible allows us to aggressively make changes to quorums–we can always revert the change if the impaired member returns. Some other systems have periodic stalls as leases expire and quorum membership needs to be re-established. Aurora neither pays the durability penalty of delaying the membership change operation until the lease expires, nor does it pay the performance penalty of delaying reads, writes, or commits while quorum membership is being established.

Aurora has made advances in a number of different areas—the approach we've taken to integrating databases and distributed systems is core to many of these. I hope you've found this set of posts about how we use quorums and avoid some of their pitfalls to be interesting and perhaps even helpful as you think about how to design your own applications and systems. The techniques we used are broadly applicable, although they do have implications across numerous elements of the stack.

If you have other questions or topics you'd like me to cover, leave a comment here or ping us at aurora-pm@amazon.com.

Migrating a SQL Server Database to a MySQL-Compatible Database Engine

Using Amazon Redshift for Fast Analytical Reports

Testing Amazon RDS for Oracle: Plotting Latency and IOPS for OLTP I/O Pattern

Get Started with Amazon Elasticsearch Service: Filter Aggregations in Kibana

Useful Documentation Links

Cloud Databases with AWS

Amazon RDS

AWS Database Migration Service

Amazon DynamoDB

Amazon ElastiCache

Amazon Redshift

AWS Blogs

AWS Blog

AWS Big Data