

您似乎是从中国境内访问我们的网站的。请导航至我们的优化版网站：[amazonaws-china.com](https://amazonaws-china.com)。

[AWS Database Blog](#)

# Amazon Aurora Under the Hood: Quorum Reads and Mutating State

by Anurag Gupta | on 16 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.*

In [my last post](#), I talked about the benefits of using a quorum model. I discussed how such systems are resilient in the face of latency outliers, short periods of unavailability, and the long-term loss of disks and nodes. That raises the obvious question—if quorums are so awesome, why doesn't everyone use them?

## Reads become slow in quorum systems

One issue is that reads become slow in quorum systems. Quorum models require that there be at least one member in both the read quorum and write quorum. In a system like [Amazon Aurora](#) with a quorum of six, that means that you'd need to read three copies of data to ensure you overlapped with a write quorum of four.

That's unfortunate. Usually when you're reading a database page, it means that you've missed in the buffer cache, and your SQL statement is blocked waiting for the I/O before it can proceed. To read three copies of data, you want to try to access circa five, to mask outlier latency and intermittent availability issues. Doing that puts burden on the network—database pages are fairly large and the read amplification is meaningful. The performance of quorum reads doesn't compare well to a traditional replication system, where data is written to all copies but can be read from any single one of them.

However, Aurora avoids quorum amplification during writes. We do write out six copies, but we only write log records, not full data pages. The data pages are assembled within the storage node from prior versions of the data page and the incoming log. We can also write asynchronously. Neither are possible for reads.

[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](#)

---

How do we avoid it? The key thing is to use state.

State is often considered a dirty word in distributed systems—it is hard to manage and coordinate consistent state as you scale nodes and encounters faults. Of course, the entire purpose of database systems is to manage state, providing atomicity, consistency, isolation, and durability (ACID). Aurora sits at the intersection of these two technology domains. Much of our innovation has come from applying concepts from one domain to drive progress in the other.

Although it is difficult to establish consensus on distributed state without communication, there are often local oases of consistency that you can use to avoid the need for consensus, coordination, or locking. The specific example we apply here is that of read views. Many database systems have a similar concept, but let's focus on MySQL.

MySQL, like all [relational databases](#), needs to provide ACID support. A read view establishes a logical point in time, before which a SQL statement must see all changes that have been committed and not see any changes that are not yet committed. MySQL does this by establishing the log sequence number (LSN) of the most recent commit. This approach ensures that all changes already committed are to be made visible, and uses an active transactions list to establish the list of changes that should not be seen. When a statement with a particular read view inspects a data page, it needs to back out any changes for transactions that were active at the time it established a read view. This is so even if these changes are currently committed, and also affects any transactions that were started after the read-point commit LSN. When a transaction establishes a read view, it can isolate itself from all other changes going on in the system—if it can backtrack to a suitably consistent point in time.

What does this have to do with read quorums? Everything. The database is continually writing changes to storage nodes. It marks an individual change as durable once it receives four acknowledgements back. It updates the point of volume durable when all changes before that point have been registered as individually durable. As part of the bookkeeping it needs to perform to do this work, it knows which storage nodes have acknowledged which write requests and can be queried to see those changes. When a read request comes in, the request has a read-point commit LSN that the database needs to see. The database can simply dispatch the request to any storage node it knows to be complete at or beyond the read-point commit LSN.

This approach uses the bookkeeping state that we have to maintain anyway to avoid a quorum read. Instead, we read from a node that we know has the data version we need. This approach avoids considerable network, storage node, and database node processing.

### **How to avoid latency**


However, by avoiding the read quorum, we make ourselves subject to latency

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

---

latency node, and occasionally also query one of the others to ensure our latency information stays up to date.

This work is straightforward for a single database node, because it sees all writes and can easily coordinate all reads. It's more complex when you consider read replicas. In Aurora, read replicas share the same storage volume as the master database node and are asynchronously shipped the master's redo log stream to update data pages in cache. This approach not only lowers costs, but also ensures that replicas can be promoted to master nodes without loss of data or the write latency penalty of synchronous replication. Any data change that was marked as committed by acknowledgements to the write master node is durable even if that change hasn't yet propagated to a replica. These replica nodes issue their own reads, but they don't have visibility to the writes and acknowledgements to know what to read.

So, as we ship redo records from master to replica, we also ship the conceptual equivalent of a read view. This view advances the commit LSN and the information on which segments are durable to which LSN points. Generally, we're able to advance the commit LSN every 10 milliseconds or so, keeping the replicas closely aligned with the write master node with minimal coordination.

### **Avoiding destructive writes**

Key to this approach is avoiding destructive writes. A large part of the reason people need to coordinate traffic between readers and writers is to ensure that the data they need to read is visible. With read views, this coordination is greatly reduced—as long as you can revert to previous images of a page. In Aurora, we write data pages out of place. We garbage-collect old versions when we know that they are backed up and all readers have advanced their read point beyond that version. This approach allows replica nodes operating a few milliseconds behind the mastering node to establish a structurally consistent view of the database.

A relational database, at core, is a redo log that always advances, even to apply the rollback of a transaction. The data pages that comprise the database are really just point-in-time cached instantiations of the application of the redo log. The fact that most databases destructively write data pages is really just a historical curiosity based on the high cost of disks when relational databases were first created.

In Aurora, using the approach outlined above, we don't really have read quorums. Instead, we have repair quorums. We only need to query a read quorum when we lose the cached state at the write master database node. If we have to restart the master instance or promote a replica to master, we do need to query at least a read quorum to rebuild our local state. It turns out that we need to do so anyway to understand what transactions are committed—likely we'll discuss that in another blog post someday. Given that database

[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](#)

## Summary

With this post and [the prior one](#), we've seen how you can use quorums to provide availability and how you can avoid the traditional read quorum overhead. In the next post, we'll talk about how to make quorum systems affordable. If you have other questions, leave a comment here or ping us at [aurora-pm@amazon.com](mailto:aurora-pm@amazon.com).

**Read Next:** [Amazon Aurora Under the Hood: Reducing Costs Using Quorum Sets](#)

 [View  
Comments](#)

## Related Posts

[Now Available – Amazon Aurora with PostgreSQL Compatibility](#)

[Automating Cross-Region and Cross-Account Snapshot Copies with the Snapshot Tool for Amazon Aurora](#)

[Migrating a SQL Server Database to a MySQL-Compatible Database Engine](#)

[Categorizing and Prioritizing a Large-Scale Move to an Open Source Database](#)

[Monitoring Amazon Aurora Audit Events with Amazon CloudWatch](#)

[Capturing Data Changes in Amazon Aurora Using AWS Lambda](#)

[Picking Your First Database to Migrate to Amazon RDS or Amazon Aurora with PostgreSQL-compatibility](#)

[Amazon Aurora Fast Database Cloning](#)