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

Alexandre Verbitski, Anurag Gupta, Debanjan Saha, James Corey, Kamal Gupta
Murali Brahmadesam, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice
Tengiz Kharatishvilli, Xiaofeng Bao
Amazon Web Services

ABSTRACT

Amazon Aurora is a high-throughput cloud-native relational database offered as part of Amazon Web Services (AWS). One of the more novel differences between Aurora and other relational databases is how it pushes redo processing to a multi-tenant scale-out storage service, purpose-built for Aurora. Doing so reduces networking traffic, avoids checkpoints and crash recovery, enables failovers to replicas without loss of data, and enables fault-tolerant storage that heals without database involvement. Traditional implementations that leverage distributed storage would use distributed consensus algorithms for commits, reads, replication, and membership changes and amplify cost of underlying storage. In this paper, we describe how Aurora avoids distributed consensus under most circumstances by establishing invariants and leveraging local transient state. Doing so improves performance, reduces variability, and lowers costs.

KEYWORDS

Databases; Distributed Systems; Log Processing; Quorum Models; Fault tolerance; Quorum Sets; Replication; Recovery; Performance

ACM Reference Format:
Alexandre Verbitski, Anurag Gupta, Debanjan Saha, James Corey, Kamal Gupta, Murali Brahmadesam, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, and Tengiz Kharatishvilli, Xiaofeng Bao. 2018. Amazon Aurora: On Avoiding Distributed Consensus for I/Os, Commits, and Membership Changes. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3183713.3196937

1 INTRODUCTION

IT workloads are increasingly moving to public cloud providers such as AWS. Many of these workloads require a relational database. Amazon Relational Database Service (RDS) provides a managed service that automates database provisioning, operating system and database patching, backup, point-in-time restore, storage and compute scaling, instance health monitoring, failover, and other capabilities. Our experience managing hundreds of thousands of database instances in RDS led to the design requirements for Aurora, a high-throughput cloud-native relational database.

In our earlier paper [12], we provided an overview of the design considerations behind Aurora. A key contribution of that paper is to show that, on a fleet-wide basis, it is insufficient to treat failures as independent. At a minimum, it is necessary to consider the correlated impact of the largest unit of failure in addition to the background noise of on-going independent failures. In AWS, the largest unit of failure a system may need to tolerate is an Availability Zone (AZ). An AZ is a subset of a Region that is connected to other AZs through low-latency networking links, but is isolated for most faults, including power, networking, software deployments, flooding, and other phenomena. Aurora supports "AZ+1" failures, resulting in six copies of data, spread across three AZs, a 4/6 write quorum, and a 3/6 read quorum as illustrated in Figure 1. Aurora implements quorum membership changes to handle unexpected failures, heat management, as well as planned software upgrades.

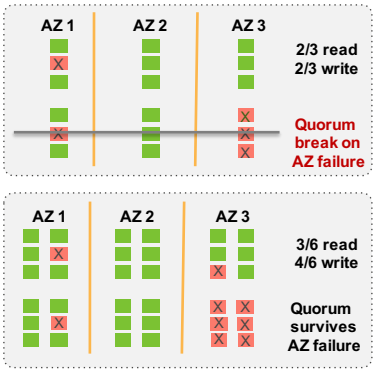

Figure 1: Why are 6 copies necessary ?

Quorum models, such as the one used by Aurora, are rarely used in high-performance relational databases, despite the benefits they provide for availability, durability, and the reduction of latency jitter. We believe this is because the underlying distributed algorithms typically used in these systems – two-phase commit (2PC), Paxos commit, Paxos membership changes, and their variants – can be expensive and incur additional network overheads. The commercial systems we have seen built on these algorithms may scale well but have order-of-magnitude worse cost, performance, and peak to average latency than a traditional relational database running on a single node against local disk.

SIGMOD'18, June 10–15, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3196937

In this paper, we show how 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. We limit our discussion to single-writer databases with read replicas. The approach described below is extensible to multi-writer databases by ordering writes at database nodes, storage nodes, and using a journal to order operations that span multiple database instances and multiple storage nodes. We describe the following contributions:

(1) How Aurora performs writes using asynchronous flows, establishes local consistency points, uses consistency points for commit processing, and re-establishes them upon crash recovery. (Section 2)
(2) How Aurora avoids quorum reads and how reads are scaled across replicas. (Section 3)
(3) How Aurora uses quorum sets and epochs to make non-blocking reversible membership changes to process failures, grow storage, and reduce costs. (Section 4)

Finally, we briefly survey related work in Section 5 and present concluding remarks in Section 6.

2 MAKING WRITES EFFICIENT

In this section, we review the Aurora storage architecture, how storage is distributed, and our quorum model. We next describe the writes performed by Aurora database instances, and how writes are batched to storage nodes. We then describe how we maintain and advance consistency points across distributed storage and how we re-establish consistency upon crash recovery.

2.1 Aurora System Architecture

Aurora uses a service-oriented architecture where database instances are loosely coupled with a multi-tenant scale-out storage service that abstracts a segmented redo log. Each database instance acts as a SQL endpoint and includes most of the components of a traditional database kernel (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 our storage fleet.

Aurora uses a quorum model, where the database reads from and writes to a subset of copies of data. Formally, a quorum system that employs V copies must obey two rules. First, the read set, V_r, and the write set, V_w, must overlap on at least one copy. This ensures a data item is not read and written by two transactions concurrently and the read quorum contains at least one site with the newest version of the data item. Second, the write set must overlap with prior write sets, which can be done by ensuring that $V_w > V/2$. This ensures two write operations from two transactions cannot occur concurrently on the same data item.

Aurora storage is partitioned into segments that individually store the redo log for their portion of the database volume as well as coalesced data blocks. The activities on the storage node are shown in more detail in Figure 2. Foreground activity in a storage node consists of (1) receiving redo records, (2) writing them to an update queue, and acknowledging them back. In background, the storage node (3) sorts and groups records, (4) gossips with peers to fill in missing records, (5) coalesces them into data blocks, (6) backs them up to Amazon Simple Storage Service (S3), (7) garbage collects backed-up data that will no longer be referenced by an instance, and (8) periodically scrubs data to ensure checksums continue to match the data on disk.

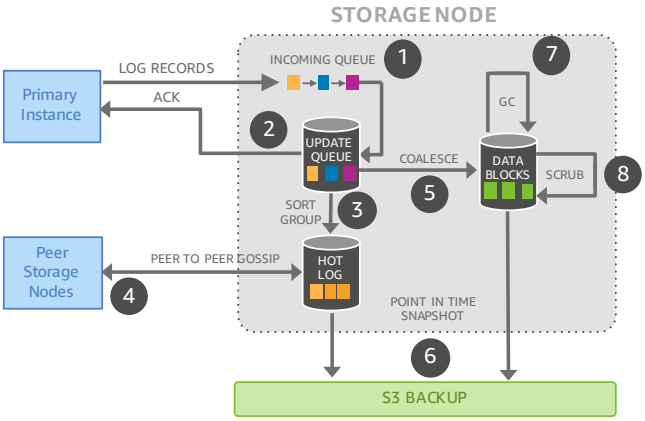

Figure 2: Activity in Aurora Storage Nodes

Segments in Aurora are the minimum unit of failure, with faults monitored and repaired automatically as part of the service. Segments are small, currently representing no more than 10GB of addressable data blocks in the database volume. Segments are replicated into protection groups, using $V = 6$, $V_w = 4$, and $V_r = 3$. These six copies are spread across three AZs, with two copies in each of the three AZs. Assuming a 10 second window to detect and repair a segment failure, it would require two independent segment failures as well as an AZ failure in the same 10 second period to lose the ability to repair a quorum. This may seem overly conservative. We don't think so. AZ failures are a correlated failure of two members in each and every quorum. Across a large fleet, some small number of quorums will be degraded, with some quorum member already failed at the time of an AZ failure. The time it takes to repair the failure of this quorum member is the time a database is vulnerable to loss of data with one additional fault.

Protection groups are concatenated together to form a storage volume, which has a one to one relationship with the database instance. While the redo log is segmented and spread across storage nodes, the Log Sequence Number (LSN) space is common across the database volume, monotonically increasing, and allocated by the database instance. This is the key invariant that allows Aurora to avoid distributed consensus for most operations.

2.2 Writes in Aurora

In Aurora, the only writes that cross the network from the database instance to the storage node are redo log records. No data blocks are written from the database instance, not for background writes, not for checkpointing, and not for cache eviction. Instead, redo log application code is run within the storage nodes, materializing blocks in background or on-demand to satisfy a read request.

Changes to data blocks modify the image in the Aurora buffer cache and add the corresponding redo record to a log buffer. These are periodically flushed to a storage driver to be made durable. Inside the driver, they are shuffled to individual write buffers for each storage node storing segments for the data volume. The driver asynchronously issues writes, receives acknowledgments, and establishes consistency points.

Each log record stores the LSN of the preceding log record in the volume, the previous LSN for the segment, and the previous LSN for the block being modified. The block chain is used by the storage node to materialize individual blocks on demand. The segment chain is used by each storage node to identify records that it has not received and fill in these holes by gossiping with other storage nodes. The full log chain is not needed by an individual storage node but provides a fallback path to regenerate storage volume metadata in case of a disastrous loss of metadata state.

Many database systems boxcar redo log writes to improve throughput. There is a challenge in deciding, with each record, whether to issue the write, to improve latency, or to wait for subsequent records, to improve write efficiency and throughput. Waiting creates performance jitter since early requests entering the boxcar have to wait for later requests or a timeout to fill the request. Jitter is greatest under low load when the boxcar times out.

In Aurora, there are many segments partitioning the redo log and the opportunity to boxcar are lower than with a single unsegmented redo log. Aurora handles this by submitting the asynchronous network operation when it receives the first redo log record in the boxcar but continuing to fill the buffer until the network operation executes. This ensures requests are sent without boxcar latency and jitter while packing records together to minimize network packets.

In Aurora, all log writes, including those for commit redo log records, are sent asynchronously to storage nodes, processed asynchronously at the storage node, and asynchronously acknowledged back to the database instance.

2.3 Storage Consistency Points and Commits

A traditional relational database working with local disk would write a commit redo log record, boxcar commits together using group commit, and flush the log to ensure that it has been made durable. When working with remote storage, it might use a two-phase commit, or a Paxos commit, or variant, to establish a consistency point since there is no individual flush operation across all storage nodes. This is heavyweight and introduces stalls and jitter into the write path. Distributed commit protocols also have failure modalities different from those of quorum writes, making it complex to reason about availability and durability.

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. More precisely, SCL is the inclusive upper bound on log records continuously linked through the segment chain without gaps. SCL is used by storage nodes as a compact way to identify missing writes when gossiping with their peers in a protection group. Note since any given write may be lost for any reason we need to tolerate missing writes in the storage nodes.

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.

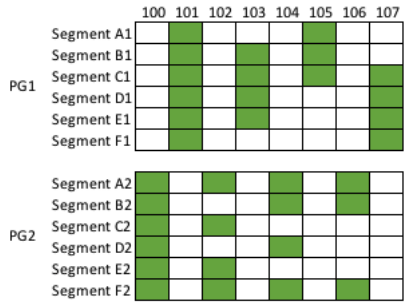

Figure 3: Storage Consistency Points

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.

This is possible because storage nodes do not have a vote in determining whether to accept a write, they must do so. Locking, transaction management, deadlocks, constraints, and other conditions that influence whether an operation may proceed are all resolved at the database tier. Processing offloaded to the Aurora storage nodes can progress by executing idempotent operations using local state. This also ensures that failed storage nodes can transparently be repaired without involving the database instance.

A commit is acknowledged by the database to its caller once it is able to affirm that all data modified by the transaction has been durably recorded. A simple way to do so is to ensure that the commit redo record for the transaction, or System Commit Number (SCN), is below VCL. No flush, consensus, or grouping is required.

Aurora must wait to acknowledge commits until it is able to advance VCL beyond the requesting SCN. Typically, this would

require stalling the worker thread acting upon the user request. In Aurora, user sessions are multiplexed to worker threads as requests are received. When a commit is received, the worker thread writes the commit record, puts the transaction on a commit queue, and returns to a common task queue to find the next request to be processed. When a driver thread advances VCL, it wakes up a dedicated commit thread that scans the commit queue for SCNs below the new VCL and sends acknowledgements to the clients waiting for commit. There is no induced latency from group commits and no idle time for worker threads.

2.4 Crash Recovery in Aurora

Aurora is able to avoid distributed consensus during writes and commits by managing consistency points in the database instance rather than establishing consistency across multiple storage nodes. But, instances fail. Customers shut them down, resize them, and restore them to older points in time. The time we save in the normal forward processing of commits using local transient state must be paid back by re-establishing consistency upon crash recovery. This is a trade worth making since commits are many orders of magnitude more common than crashes. Since instance state is ephemeral, the Aurora database instance must be able to construct PGCLs and VCL from local SCL state at storage nodes.

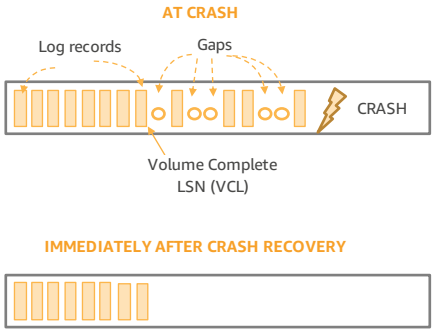

Figure 4: Log truncation during crash recovery

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. There may be a ragged edge of updates in particular segments past this point that did not yet meet quorum. These represent partial writes that did not complete and would not have been acknowledged to clients of the database. The database snips off the ragged edge of the log by recording a truncation range that annuls any log records beyond the newly computed VCL (Figure 4). This ensures that, even if in-flight asynchronous operations complete during the process of crash recovery, they are ignored. New redo records after crash recovery are allocated LSNs above the truncation range.

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 to establish short term entitlements to access the system, but leases introduce latency when one needs to wait for expiry. Aurora, rather than waiting for a lease to expire, just changes the locks on the door.

No redo replay is required as part of crash recovery since segments are able to generate data blocks on their own. Undo of previously active transactions is required but can occur after the database has been opened in parallel with user activity.

3 MAKING READS EFFICIENT

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.

In a quorum system, the I/O required for a read is amplified by the size of the read quorum. Network traffic is far higher since one is reading full data blocks, unlike writes, where Aurora only ships log records. A buffer cache miss in Aurora's quorum model would seem to require a minimum of three read I/Os, and likely five, to mask outlier latency and intermittent unavailability. Read performance in quorum systems compares poorly to traditional replication models where one writes to all copies, enabling a read from just one, though those models have worse write availability.

3.1 Avoiding quorum reads

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 MySQL does this by establishing the most recent SCN and a list of transactions active as of that LSN. Data blocks seen by a read request must be at or after the read view LSN and back out any transactions either active as of that LSN or started after that LSN. Aurora PostgreSQL also uses MVCC, though writes records out of place, recording the transaction id with each record, and vacuuming old versions periodically. Snapshot isolation is straightforward in a single-node database instance by having a transaction read the last durable version of a database block and apply undo to rollback any changes. One must apply an invariant that undo records may not be purged until all read views have advanced.

Even though Aurora does not write blocks to storage from the database instance, it must support write-ahead logging by ensuring redo log records for dirty blocks have been made durable before discarding the block from cache. This ensures that the latest version of a data block can always be found either in cache or in the cache

or by finding the latest durable version of the block in one of the segments of the protection group that it belongs to.

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. Avoiding the amplification of read quorums does make Aurora subject to latency when storage nodes are down or jitter when they are busy. We manage this by tracking response time from storage nodes for read requests. The database instance will usually issue a request to the segment with the lowest measured latency, but occasionally also query one of the others in parallel to ensure up to date read latency response times. If a request is taking longer than expected, will issue a read to another storage node and accept whichever one returns first. This caps the latency due to slow or unavailable segments. In an active system, this can be done without request timeouts by inspecting the list of outstanding requests when performing other I/Os.

3.2 Scaling Reads Using Read Replicas

Many database systems scale reads by replicating updates from a writer instance to a set of read replica instances. Typically, this involves transporting either logical statement updates or physical redo log records from the writer to the readers. Replication is done synchronously if the replicas are intended as failover targets without data loss and asynchronously if replica lag or data loss during failover is acceptable.

Both synchronous and asynchronous replication have undesirable characteristics. Synchronous replication introduces performance jitter and failure modalities in the write path. Asynchronous replication introduces data loss on failure of the writer. In both cases, replication takes time to set up, requiring copying the underlying database volume and catching up on active changes. It is also expensive, since it doubles not only the instance costs, but also storage costs. Much of the throughput of the replica instance goes to replicate write activity, not to scaling reads.

Aurora supports logical replication to communicate with non-Aurora systems and in cases where the application does not want physical consistency – for example, when schemas differ. Internally, within an Aurora cluster, we use physical 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.

This approach allows Aurora customers to quickly set up and tear down replicas in response to sharp demand spikes, since durable state is shared. Adding replicas does not change availability or durability characteristics, since durable state is independent from the number of instances accessing that state. There is little latency added to the write path on the writer instance since replication is asynchronous. Since we only update cached data blocks on the replicas, most resources on the replica remain available for read requests. And most importantly, if a commit has been marked durable and acknowledged to the client, there is no data loss when a replica is promoted to a write instance – it only needs to run a local crash recovery to align its in-memory state.

3.3 Structural Consistency in Aurora Replicas

Managing structural consistency with asynchronous operations against shared durable state requires care. A single writer has local state for all writes and can easily coordinate snapshot isolation, consistency points for storage, transaction ordering, and structural atomicity. It is more complex for replicas.

Aurora uses three invariants to manage 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.

To understand structural consistency on the replica, let us first examine structural consistency on the writer instance, using Aurora MySQL as an example. Each database transaction in Aurora MySQL is a sequence of ordered mini-transactions (MTRs) that are performed atomically. Each MTR is composed of changes to one or more data blocks, represented as a batch of sequenced redo log records to provide consistency of structural changes, such as those involving B-Tree splits. The database instance acquires latches for each data block, allocates a batch of contiguously ordered LSNs, generates the log records, issues a write, shards then into write buffers for each protection group associated with the blocks, and writes them to the various storage nodes for the segments in the protection group. We use an additional consistency point, the Volume Durable LSN (VDL), to represent the last LSN below VCL representing an MTR completion.

Replicas do not have the benefit of the latching used at the writer instance to prevent read requests from seeing non-atomic structural updates. To create equivalent ordering, we ensure that log records are only shipped from the writer instance in MTR chunks. At the replica, they must be applied in LSN order, applied only if above the VDL in the writer as seen in the replica, and applied atomically in MTR chunks to the subset of blocks in the cache. Read requests are made relative to VDL points to avoid seeing structurally inconsistent data.

3.4 Snapshot Isolation and Read View Anchors in Aurora Replicas

Once we have ensured that cached replica state is structurally consistent, allowing traversal of physical data structures, we must also ensure it is also logically consistent using snapshot isolation.

The redo log seen by a read replica does not carry the state needed to establish SCL, PGCL, VCL, or VDL consistency points. Nor is the read replica in the communication path between the writer and storage nodes to establish this state on its own. Note that VDL advances based on acknowledgements from storage nodes, not redo issuance from the writer. The writer instance sends VDL

update control records as part of its replication stream. Although the active transaction list can be reconstructed at the replica using redo records and VDL advancement, for efficiency reasons we ship commit notifications and maintain transaction commit history. Read views at the replica are built based on these VDL points and transaction commit history. Replicas revert active transactions for MVCC using undo, just as on the writer instance.

Since VDL on the replica may lag the writer, Aurora storage nodes must ensure that past values are available to be read. Aurora blocks are written out-of-place and non-destructively. Older versions are not garbage collected until we can assure neither the writer instance or any replica might need to access it. We do this by maintaining a Protection Group Minimum Read Point LSN (PGMRPL), representing the lowest LSN read point for any active request on that database instance. A storage node may only advance its garbage collection point once PGMRPL has advanced for all instances that have opened the volume. The storage nodes will only accept read requests between PGMRPL and SCL.

4 FAILURES AND QUORUM MEMBERSHIP

Managing quorum failures is complex. Traditional mechanisms cause I/O stalls while membership is being changed. They are generally intolerant of additional failures during the membership change process. Most membership change protocols are intolerant of readmitting previously fenced-out members which is particularly challenging – there is considerable state on storage nodes using modern disks and repair takes time. For these reasons, systems tend to be conservative about changing membership, increasing latency and risking multiple faults that break quorum.

The probability of failures grows with the number of segments. In Aurora, with six segments spread across three AZs for every 10GB of user data, a 64TB volume has 38,400 segments. Failures of storage nodes, top of rack switches, network paths, or entire AZs can impact many database volumes at the same time and require several repairs. In this section, we describe how Aurora supports I/O processing, multiple faults, and member re-introduction while performing membership changes.

4.1 Using Quorum Sets to Change Membership

Consider a protection group with the six segments A, B, C, D, E, and F. In Aurora, the write quorum is any four members out of this set of six, and the read quorum is any three members. Let us assume that a database instance or monitoring agent stops receiving timely acknowledgements for segment F and wants to consider replacing it with a new segment G. However, F may be encountering a temporary failure and may come back quickly. It may be processing requests, but not be observable to this monitor. It may just be busy. At the same time, we do not want to wait to see if F comes back. It may be permanently down. Waiting extends the duration of impairment, during which we may see additional faults and increased latency.

Aurora uses the abstraction of quorum sets to quickly transition membership changes, using Boolean logic to ensure more sophisticated read quorum and write quorums that are guaranteed to overlap. We make at least two transitions per membership change, ensuring each transition is reversible. Each membership change to a protection group is associated with a membership epoch, which is monotonically incremented with each change. Membership changes do not block either reads or writes.

Each read or write request from an instance and each gossip request from a peer segment passes in the epoch based on their current understanding of quorum membership. As with volume epochs, clients with stale membership epochs have their requests rejected and must update membership information. An epoch increment requires a write quorum to be met, just as any other write does. The request to increment membership epoch must pass in the correct membership epoch, just as any other request does. As with our other epochs, membership epochs ensure we can update 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.

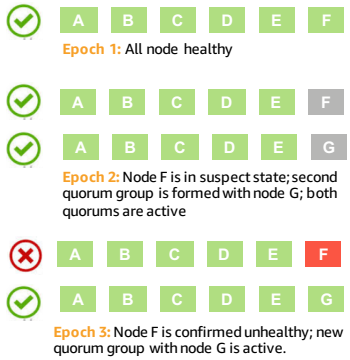

Figure 5: Quorum Membership Changes

Figure 5 illustrates how we replace segment F with segment G. Rather than attempting to directly transition from ABCDEF to ABCDEG, we make our transition in two steps. First, we add G to our 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, we can make a second membership change back to ABCDEF. That quorum subset met our write quorum and is an available next step. If F continues to be down once G has completed hydrating from its peers, we can make a membership change to ABCDEG. That quorum subset also met our write quorum and is an available next step. We do not discard any durable state until back to a fully repaired quorum.

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. Note that, both with a single failure and with multiple failures, simply writing to the four members ABCD meets quorum.

Quorum membership changes have the same failure characteristics as read and write I/Os. Using Boolean logic, we can prove that each transition is correct, safe, and reversible, whatever the

sequence of errors and repairs may be. Transitions require only the single epoch update to the write quorum of a protection group. Updates of stale state are similarly simple, requiring just one additional request past the one rejected.

We also use epochs to manage volume growth, using a volume geometry epoch that increments with each protection group added to the volume. This can also be used to change the quorum model itself, for example, when moving from a 4/6 write quorum to 3/4 to handle the extended loss of an AZ.

4.2 Using Quorum Sets to Reduce Costs

Quorums are generally thought of as a collection of like members, grouped together to transparently handle failures. However, there is nothing in the quorum model to prevent unlike members with differing latency, cost, or durability characteristics.

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 three copies of the data rather than a full six while satisfying our requirement to support AZ+1 failures.

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

Repairing a tail segment simply requires reading from the other members of the protection group, using our SCL to determine and fill in the gaps from other quorum members with SCLs higher than our own. Repairing a full segment is a bit more complex since the segment being repaired may have been the only full segment that saw the last write to the protection group.

Even so, we must have at least one other full segment from which we can read data blocks even if it has not seen the most recent write. We have enough copies of the redo log record so that we can rebuild a full segment and be up to date. We also gossip between the segments of a quorum to ensure that any missing writes are quickly filled in. This reduces the probability we need to rebuild a full segment without adding a performance burden to our write path. Once we have our full segment baseline, we can obtain redo log records from other segments using our SCL in the same manner as tail segments.

There are many options available once one moves to quorum sets of unlike members. One can combine local disks to reduce latency and remote disks for durability and availability. One can combine SSDs for performance and HDDs for cost. One can span quorums across regions to improve disaster recovery. There are numerous moving parts that one needs to get right, but the payoffs can be significant. For Aurora, the quorum set model described earlier lets us achieve storage prices comparable to low-cost alternatives, while providing high durability, availability, and performance.

5 RELATED WORK

In this section we discuss other contributions and how they relate to the techniques used in Aurora and discussed in this paper.

Consensus and Distributed Transactions. Distributed systems rely on consensus to allow a group of processes to agree on a single value and tolerate faults in one or more of its members. Some notable consensus algorithms include Paxos and variants [4, 5], Raft [9], and Viewstamped Replication [8]. A distributed database requires a commit protocol that enforces that all processes start out in a "working" state and all either end in an "aborted" or "committed" state. Distributed commit may be implemented using consensus protocols such as Paxos or other approaches like 2-phase commit and can incur considerable network overheads. Another recent system that avoids the use of distributed commit is Calvin [11] which implements a transaction scheduling and data replication layer that uses a deterministic ordering guarantee. Since all nodes reach an agreement regarding what transactions to attempt and in what order, Calvin is able to completely avoid distributed commit protocols, reducing the contention footprints of distributed transactions.

Quorums. Quorum-based approaches have been used for distributed commit protocols [10] as well as for replicating data [3].

Distributed SQL Databases. Google Cloud Spanner [1] is a SQL database on a quorum replicated system, using Multi-Paxos to establish consensus for every write providing strong consistency guarantees. Cloud Spanner enables clustering of tables to reduce the participants in distributed transactions.

Replication. Traditional database replication techniques consume a physical or logical log that represents changes made in the database and replicates these changes in a completely independent database. For example, Liu et al [6] describe how DB2 implements transactional replication from a partitioned database system by combining the physical write-ahead log from each node. Oracle uses physical replication via Data Guard [2] to provide high availability and disaster recovery. Some database systems like MySQL support logical replication [7] using command/statement logging [13].

6 CONCLUSIONS

Aurora avoids considerable network, storage, and database processing by leveraging a few simple techniques to avoid complex, brittle, and expensive consensus protocols. Most distributed consensus algorithms abhor state and establish their baseline from first principles. But, databases are all about the management of state. Why not use it for our own benefit?

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. Coordination and

consensus is rarely required. While this state is ephemeral, it can be re-established when recovering from failure.

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. This also simplifies the process of coordinating multiple request processors, as shown here for replicas operating against common storage.

Epochs provide a simple way to make changes to a distributed system, only relying on the basic notions of reading and writing to the relevant quorums. This ensures there is a consistent way to reason about availability and durability, and that there are no sharp edges when recovering from failures or changing how one must interact with a quorum. The combination of epochs and quorum sets make changes reversible and non-blocking, making membership change decisions inconsequential. Quorum sets also open up system design to more sophisticated architectures to reduce latency and cost while improving availability and durability.

We believe these techniques are broadly applicable beyond systems like Aurora to other systems coordinating multiple actors or involving shared state.

ACKNOWLEDGMENTS

We thank the entire Aurora MySQL and Aurora PostgreSQL development teams for their efforts on the project, including our current members as well as our distinguished alumni (Sam McKelvie, Yan Leshinsky, Lon Lundgren, Pradeep Madhavarapu, Hyungsoo Jung, and Stefano Stefani). We are especially grateful to Mehul Shah for his help revising the paper.

REFERENCES

[1] David F. Bacon, Nathan Bales, Nico Bruno, Brian F. Cooper, Adam Dickinson, Andrew Fikes, Campbell Fraser, Andrey Gubarev, Milind Joshi, Eugene Kogan, Alexander Lloyd, Sergey Melnik, Rajesh Rao, David Shue, Christopher Taylor, Marcel van der Holst, and Dale Woodford. 2017. Spanner: Becoming a SQL System. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. ACM, New York, NY, USA, 331–343. https://doi.org/10.1145/3035918.3056103

[2] Larry Carpenter, Joseph Meeks, Charles Kim, Bill Burke, Sonya Carothers, Joydip Kundu, Michael Smith, and Nitin Vengurlekar. 2009. *Oracle Data Guard 11G Handbook* (1 ed.). McGraw-Hill, Inc., New York, NY, USA.

[3] David K. Gifford. 1979. Weighted Voting for Replicated Data. In *Proceedings of the Seventh ACM Symposium on Operating Systems Principles (SOSP '79)*. ACM, New York, NY, USA, 150–162. https://doi.org/10.1145/800215.806583

[4] Leslie Lamport. 1998. The Part-time Parliament. *ACM Trans. Comput. Syst.* 16, 2 (May 1998), 133–169. https://doi.org/10.1145/279227.279229

[5] L. Lamport. 2001. Paxos made simple. *ACM SIGACT News* 32, 4 (2001), 18–25.

[6] Chengfei Liu, Bruce G. Lindsay, Serge Bourbonnais, Elizabeth B. Hamel, Tuong C. Truong, and Jens Stankiewitz. 2003. Capturing Global Transactions from Multiple Recovery Log Files in a Partitioned Database System. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29 (VLDB '03)*. VLDB Endowment, 987–996. http://dl.acm.org/citation.cfm?id=1315451.1315536

[7] Mike Nugent. 2010. MySQL Replication. *Linux J.* 2010, 195, Article 2 (July 2010). http://dl.acm.org/citation.cfm?id=1883478.1883480

[8] Brian M. Oki and Barbara H. Liskov. 1988. Viewstamped Replication: A New Primary Copy Method to Support Highly-Available Distributed Systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing (PODC '88)*. ACM, New York, NY, USA, 8–17. https://doi.org/10.1145/62546.62549

[9] Diego Ongaro and John Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference (USENIX ATC'14)*. USENIX Association, Berkeley, CA, USA, 305–320. http://dl.acm.org/citation.cfm?id=2643634.2643666

[10] Dale Skeen. 1982. *A Quorum-Based Commit Protocol.* Technical Report TR 82-483. Cornell University, Ithaca, NY.

[11] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. ACM, New York, NY, USA, 1–12. https://doi.org/10.1145/2213836.2213838

[12] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. ACM, New York, NY, USA, 1041–1052. https://doi.org/10.1145/3035918.3056101

[13] Chang Yao, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, and Sai Wu. 2016. Adaptive Logging: Optimizing Logging and Recovery Costs in Distributed In-memory Databases. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 1119–1134. https://doi.org/10.1145/2882903.2915208