# Amazon Aurora

November 19, 2024

Based on [VGS+17] and [VGS+18]

## 1 Introduction

Contributions:

1. How to reason about durability at cloud scale and how to design quorum systems that are resilient to correlated failures

2. How to leverage smart storage by offloading the lower quarter of a traditional database to this tier

3. How to eliminate multi-phase synchronization, crash recovery and checkpointing in distributed storage

## 2 Durability at scale

### 2.1 Replication and Correlated Failures

Goal: Tolerate 1AZ + 1 error fails

Use a quorum model with 6 voters, a write quorum of 4/6 and a read quorum of 3/6

### 2.2 Segmented Storage

To provide sufficient durability in this model, one must ensure the probability of a double fault on uncorrelated failures (Mean Time to Failure – MTTF) is sufficiently low over the time it takes to repair one of these failures (Mean Time to Repair – MTTR).

Reduce the probability of MTTF is hard, so we focus on MTTR.

We do so by partitioning the database volume into small fixed size segments, currently 10GB in size. These are each replicated 6 ways into **Protection Groups** (PGs) so that each PG consists of six 10GB segments, organized across three AZs, with two segments in each AZ. A **storage volume** is a concatenated set of PGs, physically implemented using a large fleet of storage nodes that are provisioned as virtual hosts with attached SSDs using Amazon Elastic Compute Cloud (EC2).

Segments are now our unit of independent background noise failure and repair. We monitor and automatically repair faults as part of our service. A 10GB segment can be repaired in 10 seconds on a 10Gbps network link. We would need to see two such failures in the same 10 second window plus a failure of an AZ not containing either of these two independent failures to lose quorum.

### 2.3 Operational Advantages of Resilience

For example, heat management is straightforward. We can mark one of the segments on a hot disk or node as bad, and the will be quickly repaired by migration to some other colder node in the fleet. OS and security patching is a brief unavailability event for that storage node as it is being patched. Even software upgrades to our storage fleet are managed this way.

Same as blade.

## 3 The Log is the Database

### 3.1 The Burden of Amplified Writes

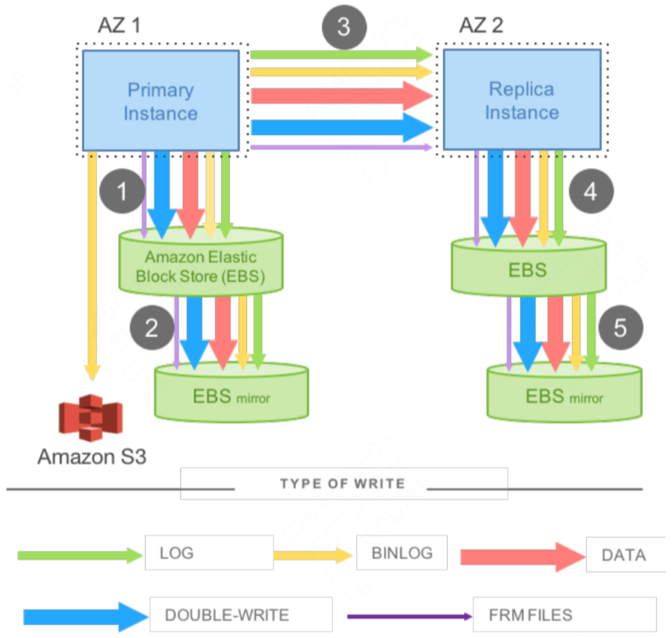


**Figure 2: Network IO in mirrored MySQL**

### 3.2 Offloading Redo Processing to Storage

In Aurora, the only writes that cross the network are redo log records. We continually materialize database pages in the background to avoid regenerating them from scratch on demand every time. And only pages with a long chain of modifications need to be rematerialized.

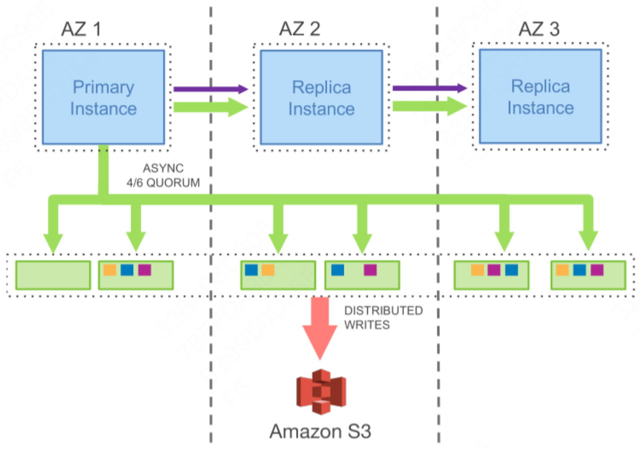


**Figure 3: Network IO in Amazon Aurora**

**Table 1: Network IOs for Aurora vs MySQL**

| **Configuration** | **Transactions** | **IOs/Transaction** |
|---|---|---|
| **Mirrored MySQL** | 780,000 | 7.4 |
| **Aurora with Replicas** | 27,378,000 | 0.95 |

In Aurora, durable redo record application happens at the storage tier, continuously, asynchronously, and distributed across the fleet. Any read request for a data page may require some redo records to be applied if the page is not current. As a result, the process of crash recovery is spread across all normal foreground processing. Nothing is required at database startup.

## 3.3 Storage Service Design Points

A core design tenet for our storage service is to minimize the latency of the foreground write request. We move the majority of storage processing to the background.

For example, it isn't necessary to run garbage collection (GC) of old page versions when the storage node is busy processing foreground write requests unless the disk is approaching capacity.

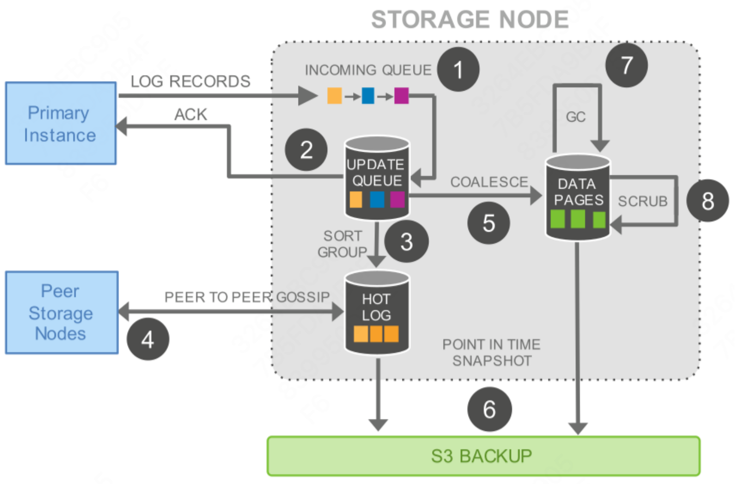


**Figure 4: IO Traffic in Aurora Storage Nodes**

1. receive log record and add to an in-memory queue
2. persist record on disk and acknowledge
3. organize records and identify gaps in the log since some batches may be lost
4. gossip with peers to fill in gaps
5. coalesce log records into new data pages,
6. periodically stage log and new pages to S3
7. periodically garbage collect old versions
8. periodically validate CRC codes on pages.

Only 1 and 2 are in the foreground path potentially impacting latency.

## 4 The Log Marches Forward

In this section, we describe how the log is generated from the database engine so that the durable state, the runtime state, and the replica state are always consistent. In particular, we will describe how consistency is implemented efficiently without an expensive 2PC protocol.

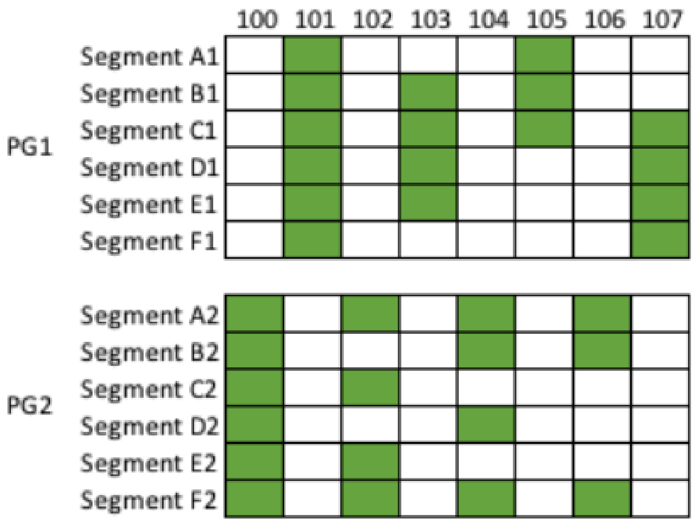
### 4.1 Asynchronous Processing

At a high level, we maintain points of consistency and durability, and continually advance these points as we receive acknowledgements for outstanding storage requests.

The database may have multiple outstanding isolated transactions, which can complete (reach a finished and durable state) in different order than initiated. Supposing the database crashes or reboots, the determination of whether to roll back is separate for each of these individual transactions.

Upon restart, before the database is allowed to access the storage volume, the storage service does its own recovery. The storage service determines the highest LSN for which it can guarantee availability of all prior log records (this is known as the VCL or Volume Complete LSN). {Note that each segment may contain holes, so VCL means for each of the log in the Volumn before VCL, there are at least four segment contains the log}

Define PGCL to be highest complete LSN in PG and consider following example where PG1 only stores odd LSN and PG2 only stores even LSN:



PGCL1 is 103, PGCL2 is 104 and VCL is 104.

During storage recovery, every log record with an LSN larger than the VCL must be truncated. The database can, however, further constrain a subset of points that are allowable for truncation by tagging log records and identifying them as CPLs or Consistency Point LSNs. We therefore define VDL or the Volume Durable LSN as the highest CPL that is smaller than or equal to VCL and truncate all log records with LSN greater than the VDL. For example, even if we have the complete data up to LSN 1007, the database may have declared that only 900, 1000, and 1100 are CPLs, in which case, we must truncate at 1000. We are *complete* to 1007, but only *durable* to 1000.

In practice, the database and storage interact as follows:

1. Each database-level transaction is broken up into multiple mini-transactions (MTRs) that are ordered and must be performed atomically.

2. Each mini-transaction is composed of multiple contiguous log records (as many as needed).

3. The final log record in a mini-transaction is a CPL.

## 4.2 Normal Operation

### 4.2.1 Writes

In Aurora, the database continuously interacts with the storage service and maintains state to establish quorum, advance volume durability, and register transactions as committed.

In the normal/forward path, as the database receives acknowledgements to establish the write quorum for each batch of log records, it advances the current VDL.

The database allocates a unique ordered LSN for each log record subject to a constraint that no LSN is allocated with a value that is greater than the sum of the current VDL and a constant called the LSN Allocation Limit (LAL) (currently set to 10 million).

Each segment of each PG only sees a subset of log records in the volume that affect the pages residing on that segment. Each log record contains a backlink that identifies the previous log record for that PG. These backlinks can be used to track the point of completeness of the log records that have reached each segment to establish a **Segment Complete LSN** (SCL) that identifies the greatest LSN below which all log records of the PG have been received.

### 4.2.2 Commits

Transaction commits are completed asynchronously.

When a client commits a transaction, the thread handling the commit request sets the transaction aside by recording its "commit LSN" as part of a separate list of transactions waiting on commit and moves on to perform other work.

The equivalent to the WAL protocol is based on completing a commit, if and only if, the latest VDL is greater than or equal to the transaction's commit LSN. As the VDL advances, the database identifies qualifying transactions that are waiting to be committed and uses a dedicated thread to send commit acknowledgements to waiting clients. Worker threads do not pause for commits, they simply pull other pending requests and continue processing.

### 4.2.3 Reads

Aurora ensures that a page in the buffer cache is alwas of the latest version. The guarantee is implemented by evicting a page from the cache only if its "page LSN" (identifying the log record associated with the latest change to the page) is greater than or equal to the VDL. This protocol ensures taht:

1. all changes in the page have been hardened in the log
2. on a cache miss, it is sufficient to request a version of the page as of the current VDL to get its latest durable version.

The database does not need to establish consensus using a read quorum under normal circumstances. When reading a page from disk, the database establishes a read-point, representing the VDL at the time the request was issued. The database can then select a storage node that is complete with respect to the read point, knowing that it will therefore receive an up to date version. A page that is returned by the storage node must be consistent with the expected semantics of a mini-transaction (MTR) in the database. Since the database directly manages feeding log records to storage nodes and tracking progress (i.e., the SCL of each segment), it normally knows which segment is capable of satisfying a read (the segments whose SCL is greater than read-point) and thus can issue a read request directly to a segment that has sufficient data.

Given that the database is aware of all outstanding reads, it can compute at any time the Minimum Read Point LSN on a per-PG basis. If there are read replicas the writer gossips with them to establish the per-PG Minimum Read Point LSN across all nodes. This value is called the **Protection Group Min Read Point LSN** (**PGMRPL**) and represents the "low water mark" below which all the log records of the PG are unnecessary. In other words, a storage node segment is guaranteed that there will be no read page requests with a read-point that is lower than the PGMRPL. Each storage node is aware of the PGMRPL from the database and can, therefore, advance the materialized pages on disk by coalescing the older log records and then safely garbage collecting them.

### 4.2.4 Replicas

In Aurora, a single writer and up to 15 read replicas can all mount a single shared storage volume. The log stream generated by the writer and sent to the storage nodes is also sent to all read replicas.

In the reader, the database consumes this log stream by considering each log record in turn. If the log record refers to a page in the reader's buffer cache, it uses the log applicator to apply the specified redo operation to the page in the cache. Otherwise it simply discards the log record.

The replica obeys 2 rules:

1. the only log records that will be applied are those whose LSN is less than or equal to the VDL

2. the log records that are part of a single mini-transaction are applied atomically in the replica's cache to ensure that the replica sees a consistent view of all databases objects.

## 4.3 Recovery

On recovery, the database contacts for each PG, a read quorum of segments which is sufficient to guarantee discovery of any data that could have reached a write quorum. Once the database has established a read quorum for every PG it can recalculate the VDL above which data is truncated by generating a truncation range that annuls every log record after the new VDL, up to and including an end LSN which the database can prove is at least as high as the highest possible outstanding log record that could ever have been seen. The database infers this upper bound because it allocates LSNs, and limits how far allocation can occur above VDL (the 10 million limit described earlier).

The database still needs to perform undo recovery to unwind the operations of in-flight transactions at the time of the crash. However, undo recovery can happen when the database is online after the system builds the list of these in-flight transactions from the undo segments.

## 5 Lessons Learned

### 5.1 Multi-tenancy and database consolidation

## 6 Problems

## 7 References

nice blog

## References

[VGS+17] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. Amazon aurora: Design considerations for high throughput cloud-native relational databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1041–1052, 2017.

[VGS+18] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, James Corey, Kamal Gupta, Murali Brahmadesam, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvilli, et al. Amazon aurora: On avoiding distributed consensus for i/os, commits, and membership changes. In *Proceedings of the 2018 International Conference on Management of Data*, pages 789–796, 2018.