Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases

DOR SHWARTZ
Agenda:

1. Overview
2. Durability at scale.
3. The log is the DB
4. The Log Marches Forward
5. Putting it all together.
6. Results.
7. Demo
8. Wrap it up.
Overview:

- Relational DB service that is offered as part of AWS.
- Move constraint from CPU & Storage to **Network**.
- Advantages:
  - Building storage as an independent fault-tolerant and self-healing service across multiple data centers.
  - Only writing redo log records to storage.
  - Move critical functions to continuous asynchronous operations.
Durability at Scale

Methods to create durability:
- Decoupling storage from compute.
- Replications.

The "quorum model":
- "A transaction is executed if the majority of sites vote to execute it."

\[
\begin{align*}
V_r + V_w &> V \\
V_w &> V/2
\end{align*}
\]

Why 2/3 isn’t enough?
- Solution: 6 replicas across 3 AZ with 2 replicas each.
  \[V_w = 4 \text{ and } V_r = 3\]
- Allows losing a single AZ and one single node and still be able to read and losing any two nodes and maintain the ability to write.
Segmented Storage

- Reducing MTTR – portioning DB into small fixed size segments (10 GB at the moment * 6), two segments in each AZ.
- Each segment is independent in aspects of noise failure and repair.
- Results:
  - We would need to see two such failures in the safe 10 second window plus a failure of an AZ not containing either of these two independent failures to lose quorum.
  - Resilient to long and short failures.
  - Maintenance/upgrade becomes easy.
The log is the DATABASE

- The Burden of Amplified Writes
  - Traditional methods with 4/6 quorum requires many I/O operations which creates heavy PPS (packets per second) burden.
  - In addition maintaining synchronization between different nodes require heavy load as well.
  - In theory each operation requires both the operation itself and a log record for redo, in practice even more.
Offloading Redo Processing to Storage

- The only writes that cross the network are redo log records.
- Log applicator is pushed to the storage tier.
  - Pages are created in the background constantly and on demand.
- Auroras page materialization is governed by the length of the chain for a given page.

<table>
<thead>
<tr>
<th>Table 1: Network IOs for Aurora vs MySQL</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Configuration</strong></td>
</tr>
<tr>
<td>Mirrored MySQL</td>
</tr>
<tr>
<td>Aurora with Replicas</td>
</tr>
</tbody>
</table>
Offloading Redo Processing to Storage - Recovery

- In classic database systems -> return to last checkpoint and replay the log to ensure that all persisted redo records have been applied.
  
  In Aurora -> durable redo record application happens at the storage tier, continuously, asynchronously, and distributed across the fleet.
  
- The process of crash recovery is spread across all normal foreground processing.
Main goal is to minimize latency for foreground write request.
- Moving the majority of the processing to the background.
- Allows us to trade CPU for Disk.
- Distributing background activity to prevent long log queue in one node which will cause home to appear as slow node.
Storage Service Design Points - Diagram

Figure 4: IO Traffic in Aurora Storage Nodes

Let’s examine the various activities on the storage node in more detail. As seen in Figure 4, it involves the following steps: (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, and finally (8) periodically validate CRC codes on pages.

Note that not only are each of the steps above asynchronous, only steps (1) and (2) are in the foreground path potentially impacting latency.
Figure 4: IO Traffic in Aurora Storage Nodes
The Log Marches Forward

- Asynchronous Processing
  - They use the fact that the log advances as an ordered sequence of changes with LSN (Log Sequence Number).
  - Asynchronous fashion instead of using 2PC.
  - Gossip with peers in PG as a way to receive missing data.
  - Runtime state maintained allows single segment reads (no need for quorum).
  - Volume Complete LSN vs Volume Durable LSN and Consistency Point LSNs.
  - Database level transaction => multiple mini-transactions (atomically) => multiple contiguous log records. The final log record in a mini-transaction is a CPL.
<table>
<thead>
<tr>
<th>TX – DB LEVEL</th>
<th>MTR – Atomically</th>
<th>MTR</th>
<th>MTR</th>
<th>LSN – CPL</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LSN</td>
<td>LSN</td>
<td>LSN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LSN</td>
<td></td>
<td>LSN</td>
<td></td>
</tr>
<tr>
<td></td>
<td>LSN – CPL</td>
<td></td>
<td>LSN</td>
<td></td>
</tr>
</tbody>
</table>

- **LSN** – Log sequence number
- **MTR** – Mini transaction
- **CPL** – Consistency point LSN
- **VCL** – Volume complete LSN
- **VDL** – Volume durable LSN
Normal Operation 1/2

- ** Writes:**
  - Each transaction generates their own redo log records with unique LSN. LSN > VDL + LAL
  - Each log record contains a backlink that identifies the previous log record for that PG. It can be used to track the point of completeness. -> SCL = Segment Complete LSN. SCL is used in the gossip stage.

- ** Commits:**
  - Completed asynchronously.
  - A thread recording a tx “commit LSN”.
  - Completing the commit iff 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.
Normal Operation

Reads:
- Cache and victims in a regular way.
- A page in the buffer cache must always be of the latest version.
- No need for read quorum – “read-point” (representing the VDL at the time the request was issued). The DB can select to read from completed nodes.
- Protection Group Min Read Point LSN – “low water mark”. Used also to remove old logs.
- Concurrency control protocols are executed in the DB engine. Like traditional MySQL.

Replicas:
- A single writer and up to 15 read replicas can all mount a single shared storage volume.
  - Read replicas add no additional costs in terms of storage or disk write ops.
- 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. Else it discards the log record.
- The only log records that will be applied are those whose LSN is less than or equal to the VDL.
- 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 database objects.
Recovery

- Traditional DB rely on write-ahead log.
- The redo log applicator is decoupled from the database and operates on storage nodes, in parallel, and all the time in the background.
  - As a result -> very fast recovery
- The database does need to reestablish its runtime state after a crash.
- The truncation ranges are versioned with epoch numbers, and written durably to the storage service so that there is no confusion over the durability of truncations in case recovery is interrupted and restarted.
- The database still needs to perform undo recovery to unwind the operations of in-flight transactions at the time of the crash.
InnoDB for example:

- Write: data being modified in buffer pages -> redo log written to buffers of WAL in LSN order.
- Commit: WAL protocol needs only the redo logs, the buffer pages are written to disk in the background during eviction or at a checkpoint.
- In the Aurora InnoDB variant, the redo log records representing the changes that must be executed atomically in each MTR are organized into batches that are sharded by the PGs each log record belongs to, and these batches are written to the storage service.

Concurrency control is implemented entirely in the database engine without impacting the storage service.

RDS is used for the control plane. RDS contains an agent on the database instance called Host Manager for Health monitoring.
Putting It All Together 2/3

- Each database instance is part of a cluster (single geo region) that consists of a single writer and zero or more read replicas.
- AZ is spread across multiple clusters and connected to a storage fleet in the same region.
- Each DB instance can communicate on three VPC networks: the customer VPC through which customer applications interact with the engine, the RDS VPC through which the database engine and control plane interact with each other, and the Storage VPC through which the database interacts with storage services.
The storage service is deployed on a cluster of EC2 VMs that are provisioned across at least 3 AZs in each region.

They are collectively responsible for provisioning multiple customer storage volumes, reading and writing data to and from those volumes, and backing up and restoring data from and to those volumes.

The storage nodes manipulate local SSDs and interact with database engine instances, other peer storage nodes, and the backup/restore services that continuously backup changed data to S3 and restore data from S3 as needed.

DynamoDB, Amazon Simple Workflow Service, Metric collection services.
Performance Results – Industry 1/3

- Scaling with instance sizes - 1GB (250 tables), 5 EC2 instances
Performance Results - Industry 2/3

- **Throughput with varying data sizes:**

<table>
<thead>
<tr>
<th>DB Size</th>
<th>Amazon Aurora</th>
<th>MySQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 GB</td>
<td>107,000</td>
<td>8,400</td>
</tr>
<tr>
<td>10 GB</td>
<td>107,000</td>
<td>2,400</td>
</tr>
<tr>
<td>100 GB</td>
<td>101,000</td>
<td>1,500</td>
</tr>
<tr>
<td>1 TB</td>
<td>41,000</td>
<td>1,200</td>
</tr>
</tbody>
</table>

Table 2: SysBench Write-Only (writes/sec)

- **Scaling with user connections:**

<table>
<thead>
<tr>
<th>Connections</th>
<th>Amazon Aurora</th>
<th>MySQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>40,000</td>
<td>10,000</td>
</tr>
<tr>
<td>500</td>
<td>71,000</td>
<td>21,000</td>
</tr>
<tr>
<td>5,000</td>
<td>110,000</td>
<td>13,000</td>
</tr>
</tbody>
</table>

Table 3: SysBench OLTP (writes/sec)
Performance Results – Industry 3/3

Scaling with Replicas:

Table 4: Replica Lag for SysBench Write-Only (msec)

<table>
<thead>
<tr>
<th>Writes/sec</th>
<th>Amazon Aurora</th>
<th>MySQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>1,000</td>
<td>2.62</td>
<td>&lt; 1000</td>
</tr>
<tr>
<td>2,000</td>
<td>3.42</td>
<td>1000</td>
</tr>
<tr>
<td>5,000</td>
<td>3.94</td>
<td>60,000</td>
</tr>
<tr>
<td>10,000</td>
<td>5.38</td>
<td>300,000</td>
</tr>
</tbody>
</table>

Throughput with hot row contention:

Table 5: Percona TPC-C Variant (tpmC)

<table>
<thead>
<tr>
<th>Connections/Size/Warehouses</th>
<th>Amazon Aurora</th>
<th>MySQL 5.6</th>
<th>MySQL 5.7</th>
</tr>
</thead>
<tbody>
<tr>
<td>500/10GB/100</td>
<td>73,955</td>
<td>6,093</td>
<td>25,289</td>
</tr>
<tr>
<td>5000/10GB/100</td>
<td>42,181</td>
<td>1,671</td>
<td>2,592</td>
</tr>
<tr>
<td>500/100GB/1000</td>
<td>70,663</td>
<td>3,231</td>
<td>11,868</td>
</tr>
<tr>
<td>5000/100GB/1000</td>
<td>30,221</td>
<td>5,575</td>
<td>13,005</td>
</tr>
</tbody>
</table>
Performance Results – Customers 1/2

- Application response time with Aurora:

![Graph showing web application response time improvement with Aurora migration.](image)

Figure 8: Web application response time
Performance Results – Customers 2/3

- Statement Latencies with Aurora:

**Figure 9: SELECT latency (P50 vs P95)**

**Figure 10: INSERT per-record latency (P50 vs P95)**
Replica Lag with Multiple Replicas: earlier 12 minutes!!!
Lessons Learned

- Multi-tenancy and database consolidation
- Highly concurrent auto-scaling workloads
- Schema evolution
- Availability and Software Upgrades - ZeroDownTime Patch
CONCLUSION

- High throughput OLTP database with high availability and durability in a cloud-scale based environment.
- Decouple storage from compute -> the lower quarter of the database kernel was moved to an independent scalable and distributed service that managed logging and storage.
- The fundamental constraint is now the network.
- Rely on quorum models.
- Their approach has led to a simplified architecture with reduced complexity that is easy to scale as well as a foundation for future advances.