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

Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, Xiaofeng Bao

Amazon Web Services

ABSTRACT

Amazon Aurora is a relational database service for OLTP workloads offered as part of Amazon Web Services (AWS). In this paper, we describe the architecture of Aurora and the design considerations leading to that architecture. We believe the central constraint in high throughput data processing has moved from compute and storage to the network. Aurora brings a novel architecture to the relational database to address this constraint, most notably by pushing redo processing to a multi-tenant scale-out storage service, purpose-built for Aurora. We describe how doing so not only reduces network traffic, but also allows for fast crash recovery, failovers to replicas without loss of data, and fault-tolerant, self-healing storage. We then describe how Aurora achieves consensus on durable state across numerous storage nodes using an efficient asynchronous scheme, avoiding expensive and chatty recovery protocols. Finally, having operated Aurora as a production service for over 18 months, we share lessons we have learned from our customers on what modern cloud applications expect from their database tier.

Keywords

Databases; Distributed Systems; Log Processing; Quorum Models; Replication; Recovery; Performance; OLTP

1. INTRODUCTION

IT workloads are increasingly moving to public cloud providers. Significant reasons for this industry-wide transition include the ability to provision capacity on a flexible on-demand basis and to pay for this capacity using an operational expense as opposed to capital expense model. Many IT workloads require a relational OLTP database; providing equivalent or superior capabilities to on-premise databases is critical to support this secular transition.

In modern distributed cloud services, resilience and scalability are increasingly achieved by decoupling compute from storage [10][24][36][38][39] and by replicating storage across multiple nodes. Doing so lets us handle operations such as replacing misbehaving or unreachable hosts, adding replicas, failing over from a writer to a replica, scaling the size of a database instance up or down, etc.

SIGMOD'17, May 14 – 19, 2017, Chicago, IL, USA.

ACM 978-1-4503-4197-4/17/05…$15.00
DOI: http://dx.doi.org/10.1145/3035918.3056101

The I/O bottleneck faced by traditional database systems changes in this environment. Since I/Os can be spread across many nodes and many disks in a multi-tenant fleet, the individual disks and nodes are no longer hot. Instead, the bottleneck moves to the network between the database tier requesting I/Os and the storage tier that performs these I/Os. Beyond the basic bottlenecks of packets per second (PPS) and bandwidth, there is amplification of traffic since a performant database will issue writes out to the storage fleet in parallel. The performance of the outlier storage node, disk or network path can dominate response time.

Although most operations in a database can overlap with each other, there are several situations that require synchronous operations. These result in stalls and context switches. One such situation is a disk read due to a miss in the database buffer cache. A reading thread cannot continue until its read completes. A cache miss may also incur the extra penalty of evicting and flushing a dirty cache page to accommodate the new page. Background processing such as checkpointing and dirty page writing can reduce the occurrence of this penalty, but can also cause stalls, context switches and resource contention.

Transaction commits are another source of interference; a stall in committing one transaction can inhibit others from progressing. Handling commits with multi-phase synchronization protocols such as 2-phase commit (2PC) [3][4][5] is challenging in a cloud-scale distributed system. These protocols are intolerant of failure and high-scale distributed systems have a continual "background noise" of hard and soft failures. They are also high latency, as high scale systems are distributed across multiple data centers.

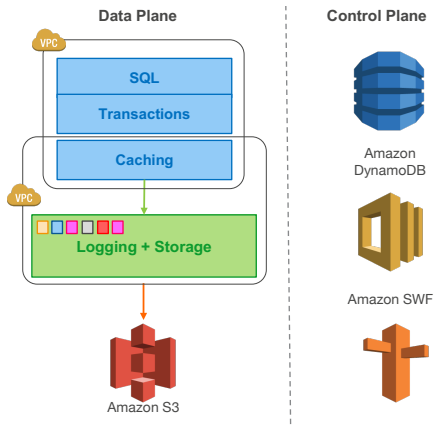

Figure 1: Move logging and storage off the database engine

In this paper, we describe Amazon Aurora, a new database service that addresses the above issues by more aggressively leveraging the redo log across a highly-distributed cloud environment. We use a novel service-oriented architecture (see Figure 1) with a multi-tenant scale-out storage service that abstracts a virtualized segmented redo log and is loosely coupled to a fleet of database instances. Although each instance still includes most of the components of a traditional kernel (query processor, transactions, locking, buffer cache, access methods and undo management) several functions (redo logging, durable storage, crash recovery, and backup/restore) are off-loaded to the storage service.

Our architecture has three significant advantages over traditional approaches. First, by building storage as an independent fault-tolerant and self-healing service across multiple data-centers, we protect the database from performance variance and transient or permanent failures at either the networking or storage tiers. We observe that a failure in durability can be modeled as a long-lasting availability event, and an availability event can be modeled as a long-lasting performance variation – a well-designed system can treat each of these uniformly [42]. Second, by only writing redo log records to storage, we are able to reduce network IOPS by an order of magnitude. Once we removed this bottleneck, we were able to aggressively optimize numerous other points of contention, obtaining significant throughput improvements over the base MySQL code base from which we started. Third, we move some of the most complex and critical functions (backup and redo recovery) from one-time expensive operations in the database engine to continuous asynchronous operations amortized across a large distributed fleet. This yields near-instant crash recovery without checkpointing as well as inexpensive backups that do not interfere with foreground processing.

In this paper, we describe three contributions:

1. How to reason about durability at cloud scale and how to design quorum systems that are resilient to correlated failures. (Section 2).
2. How to leverage smart storage by offloading the lower quarter of a traditional database to this tier. (Section 3).
3. How to eliminate multi-phase synchronization, crash recovery and checkpointing in distributed storage (Section 4).

We then show how we bring these three ideas together to design the overall architecture of Aurora in Section 5, followed by a review of our performance results in Section 6 and the lessons we have learned in Section 7. Finally, we briefly survey related work in Section 8 and present concluding remarks in Section 9.

2. DURABILITY AT SCALE

If a database system does nothing else, it must satisfy the contract that data, once written, can be read. Not all systems do. In this section, we discuss the rationale behind our quorum model, why we segment storage, and how the two, in combination, provide not only durability, availability and reduction of jitter, but also help us solve the operational issues of managing a storage fleet at scale.

2.1 Replication and Correlated Failures

Instance lifetime does not correlate well with storage lifetime. Instances fail. Customers shut them down. They resize them up and down based on load. For these reasons, it helps to decouple the storage tier from the compute tier.

Once you do so, those storage nodes and disks can also fail. They therefore must be replicated in some form to provide resiliency to failure. In a large-scale cloud environment, there is a continuous low level background noise of node, disk and network path failures. Each failure can have a different duration and a different blast radius. For example, one can have a transient lack of network availability to a node, temporary downtime on a reboot, or a permanent failure of a disk, a node, a rack, a leaf or a spine network switch, or even a data center.

One approach to tolerate failures in a replicated system is to use a quorum-based voting protocol as described in [6]. If each of the V copies of a replicated data item is assigned a vote, a read or write operation must respectively obtain a read quorum of V_r votes or a write quorum of V_w votes. To achieve consistency, the quorums must obey two rules. First, each read must be aware of the most recent write, formulated as $V_r + V_w > V$. This rule ensures the set of nodes used for a read intersects with the set of nodes used for a write and the read quorum contains at least one location with the newest version. Second, each write must be aware of the most recent write to avoid conflicting writes, formulated as $V_w > V/2$.

A common approach to tolerate the loss of a single node is to replicate data to (V = 3) nodes and rely on a write quorum of 2/3 ($V_w = 2$) and a read quorum of 2/3 ($V_r = 2$).

We believe 2/3 quorums are inadequate. To understand why, let's first understand the concept of an Availability Zone (AZ) in AWS. An AZ is a subset of a Region that is connected to other AZs in the region through low latency links but is isolated for most faults, including power, networking, software deployments, flooding, etc. Distributing data replicas across AZs ensures that typical failure modalities at scale only impact one data replica. This implies that one can simply place each of the three replicas in a different AZ, and be tolerant to large-scale events in addition to the smaller individual failures.

However, in a large storage fleet, the background noise of failures implies that, at any given moment in time, some subset of disks or nodes may have failed and are being repaired. These failures may be spread independently across nodes in each of AZ A, B and C. However, the failure of AZ C, due to a fire, roof failure, flood, etc, will break quorum for any of the replicas that concurrently have failures in AZ A or AZ B. At that point, in a 2/3 read quorum model, we will have lost two copies and will be unable to determine if the third is up to date. In other words, while the individual failures of replicas in each of the AZs are uncorrelated, the failure of an AZ is a correlated failure of all disks and nodes in that AZ. Quorums need to tolerate an AZ failure as well as concurrently occuring background noise failures.

In Aurora, we have chosen a design point of tolerating (a) losing an entire AZ and one additional node (AZ+1) without losing data, and (b) losing an entire AZ without impacting the ability to write data. We achieve this by replicating each data item 6 ways across 3 AZs with 2 copies of each item in each AZ. We use a quorum model with 6 votes (V = 6), a write quorum of 4/6 ($V_w = 4$), and a read quorum of 3/6 ($V_r = 3$). With such a model, we can (a) lose a single AZ and one additional node (a failure of 3 nodes) without losing read availability, and (b) lose any two nodes, including a single AZ failure and maintain write availability. Ensuring read quorum enables us to rebuild write quorum by adding additional replica copies.

2.2 Segmented Storage

Let's consider the question of whether AZ+1 provides sufficient durability. 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). If the probability of a double fault is sufficiently high, we may see these on an AZ failure, breaking quorum. It is difficult, past a point, to reduce the probability of MTTF on independent failures. We instead focus on reducing MTTR to shrink the window of vulnerability to a double fault. 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). The PGs that constitute a volume are allocated as the volume grows. We currently support volumes that can grow up to 64 TB on an unreplicated basis.

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. At our observed failure rates, that's sufficiently unlikely, even for the number of databases we manage for our customers.

2.3 Operational Advantages of Resilience

Once one has designed a system that is naturally resilient to long failures, it is naturally also resilient to shorter ones. A storage system that can handle the long-term loss of an AZ can also handle a brief outage due to a power event or bad software deployment requiring rollback. One that can handle a multi-second loss of availability of a member of a quorum can handle a brief period of network congestion or load on a storage node.

Since our system has a high tolerance to failures, we can leverage this for maintenance operations that cause segment unavailability. For example, **heat management** is straightforward. We can mark one of the segments on a hot disk or node as bad, and the quorum 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. We execute them one AZ at a time and ensure no more than one member of a PG is being patched simultaneously. This allows us to use agile methodologies and rapid deployments in our storage service.

3. THE LOG IS THE DATABASE

In this section, we explain why using a traditional database on a segmented replicated storage system as described in Section 2 imposes an untenable performance burden in terms of network IOs and synchronous stalls. We then explain our approach where we offload log processing to the storage service and experimentally demonstrate how our approach can dramatically reduce network IOs. Finally, we describe various techniques we use in the storage service to minimize synchronous stalls and unnecessary writes.

3.1 The Burden of Amplified Writes

Our model of segmenting a storage volume and replicating each segment 6 ways with a 4/6 write quorum gives us high resilience. Unfortunately, this model results in untenable performance for a traditional database like MySQL that generates many different actual I/Os for each application write. The high I/O volume is amplified by replication, imposing a heavy packets per second (PPS) burden. Also, the I/Os result in points of synchronization that stall pipelines and dilate latencies. While chain replication [8] and its alternatives can reduce network cost, they still suffer from synchronous stalls and additive latencies.

Let's examine how writes work in a traditional database. A system like MySQL writes data pages to objects it exposes (e.g., heap files, b-trees etc.) as well as redo log records to a write-ahead log (WAL). Each redo log record consists of the difference between the after-image and the before-image of the page that was modified. A log record can be applied to the before-image of the page to produce its after-image.

In practice, other data must also be written. For instance, consider a synchronous mirrored MySQL configuration that achieves high availability across data-centers and operates in an active-standby configuration as shown in Figure 2. There is an active MySQL instance in AZ1 with networked storage on Amazon Elastic Block Store (EBS). There is also a standby MySQL instance in AZ2, also with networked storage on EBS. The writes made to the primary EBS volume are synchronized with the standby EBS volume using software mirroring.

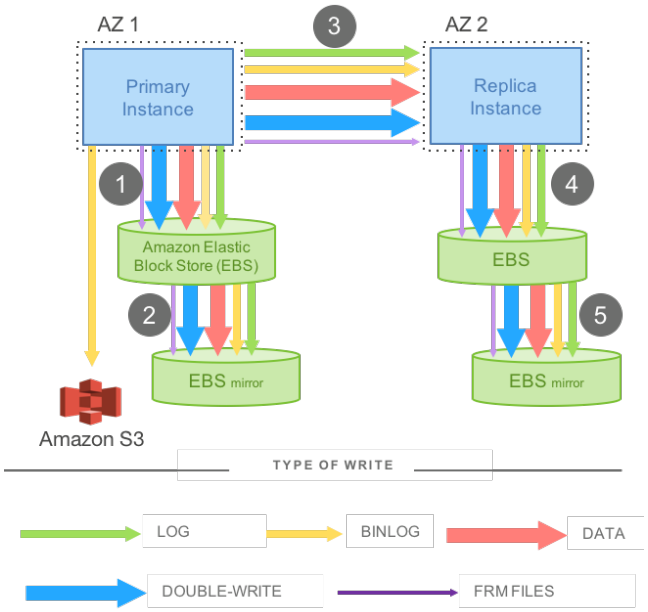

Figure 2: Network IO in mirrored MySQL

Figure 2 shows the various types of data that the engine needs to write: the redo log, the binary (statement) log that is archived to Amazon Simple Storage Service (S3) in order to support point-in-time restores, the modified data pages, a second temporary write of the data page (double-write) to prevent torn pages, and finally the metadata (FRM) files. The figure also shows the order of the actual IO flow as follows. In Steps 1 and 2, writes are issued to EBS, which in turn issues it to an AZ-local mirror, and the acknowledgement is received when both are done. Next, in Step 3, the write is staged to the standby instance using synchronous block-level software mirroring. Finally, in steps 4 and 5, writes are written to the standby EBS volume and associated mirror.

The mirrored MySQL model described above is undesirable not only because of how data is written but also because of what data is written. First, steps 1, 3, and 5 are sequential and synchronous. Latency is additive because many writes are sequential. Jitter is amplified because, even on asynchronous writes, one must wait for the slowest operation, leaving the system at the mercy of

outliers. From a distributed system perspective, this model can be viewed as having a 4/4 write quorum, and is vulnerable to failures and outlier performance. Second, user operations that are a result of OLTP applications cause many different types of writes often representing the same information in multiple ways – for example, the writes to the double write buffer in order to prevent torn pages in the storage infrastructure.

3.2 Offloading Redo Processing to Storage

When a traditional database modifies a data page, it generates a redo log record and invokes a log applicator that applies the redo log record to the in-memory before-image of the page to produce its after-image. Transaction commit requires the log to be written, but the data page write may be deferred.

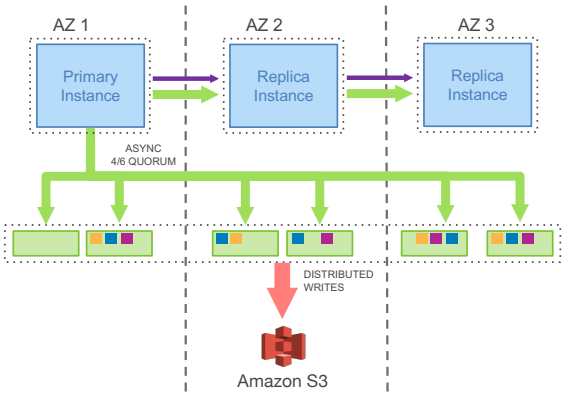

Figure 3: Network IO in Amazon Aurora

In Aurora, the only writes that cross the network are redo log records. No pages are ever written from the database tier, not for background writes, not for checkpointing, and not for cache eviction. Instead, the log applicator is pushed to the storage tier where it can be used to generate database pages in background or on demand. Of course, generating each page from the complete chain of its modifications from the beginning of time is prohibitively expensive. We therefore continually materialize database pages in the background to avoid regenerating them from scratch on demand every time. Note that background materialization is entirely optional from the perspective of correctness: as far as the engine is concerned, *the log is the database*, and any pages that the storage system materializes are simply a cache of log applications. Note also that, unlike checkpointing, only pages with a long chain of modifications need to be rematerialized. Checkpointing is governed by the length of the entire redo log chain. Aurora page materialization is governed by the length of the chain for a given page.

Our approach dramatically reduces network load despite amplifying writes for replication and provides performance as well as durability. The storage service can scale out I/Os in an embarrassingly parallel fashion without impacting write throughput of the database engine. For instance, Figure 3 shows an Aurora cluster with one primary instance and multiple replicas instances deployed across multiple AZs. In this model, the primary only writes log records to the storage service and streams those log records as well as metadata updates to the replica instances. The IO flow batches fully ordered log records based on a common destination (a logical segment, i.e., a PG) and delivers each batch to all 6 replicas where the batch is persisted on disk and the database engine waits for acknowledgements from 4 out of 6 replicas in order to satisfy the write quorum and consider the log records in question durable or *hardened*. The replicas use the redo log records to apply changes to their buffer caches.

To measure network I/O, we ran a test using the SysBench [9] write-only workload with a 100GB data set for both configurations described above: one with a synchronous mirrored MySQL configuration across multiple AZs and the other with RDS Aurora (with replicas across multiple AZs). In both instances, the test ran for 30 minutes against database engines running on an r3.8xlarge EC2 instance.

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

The results of our experiment are summarized in Table 1. Over the 30-minute period, Aurora was able to sustain 35 times more transactions than mirrored MySQL. The number of I/Os per transaction on the database node in Aurora was 7.7 times fewer than in mirrored MySQL despite amplifying writes six times with Aurora and not counting the chained replication within EBS nor the cross-AZ writes in MySQL. Each storage node sees unamplified writes, since it is only one of the six copies, resulting in 46 times fewer I/Os requiring processing at this tier. The savings we obtain by writing less data to the network allow us to aggressively replicate data for durability and availability and issue requests in parallel to minimize the impact of jitter.

Moving processing to a storage service also improves availability by minimizing crash recovery time and eliminates jitter caused by background processes such as checkpointing, background data page writing and backups.

Let's examine crash recovery. In a traditional database, after a crash the system must start from the most recent 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. 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. Given the natural variability between peak to average foreground requests from the storage tier, we have ample time to perform these tasks outside the foreground path. We also have the opportunity to trade CPU for disk. 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. In Aurora, background processing has negative correlation with foreground processing. This is unlike a traditional database, where background writes of pages and checkpointing have positive correlation with the foreground load on the system. If we build up a backlog on the system, we will throttle foreground activity to prevent a long queue buildup. Since segments are placed with high entropy across the various storage nodes in our system, throttling at one storage node is readily handled by our 4/6 quorum writes, appearing as a slow node.

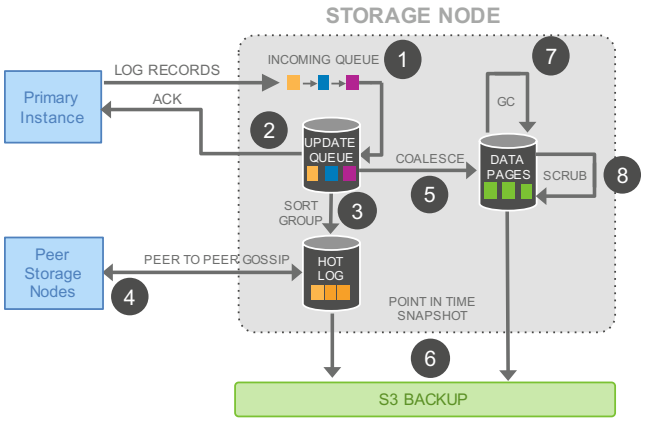

Figure 4: IO Traffic in Aurora Storage Nodes

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.

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. First, we show how we avoid expensive redo processing on crash recovery. Next, we explain normal operation and how we maintain runtime and replica state. Finally, we provide details of our recovery process.

4.1 Solution sketch: Asynchronous Processing

Since we model the database as a redo log stream (as described in Section 3), we can exploit the fact that the log advances as an ordered sequence of changes. In practice, each log record has an associated Log Sequence Number (LSN) that is a monotonically increasing value generated by the database.

This lets us simplify a consensus protocol for maintaining state by approaching the problem in an asynchronous fashion instead of using a protocol like 2PC which is chatty and intolerant of failures. At a high level, we maintain points of consistency and durability, and continually advance these points as we receive acknowledgements for outstanding storage requests. Since any individual storage node might have missed one or more log records, they gossip with the other members of their PG, looking for gaps and fill in the holes. The runtime state maintained by the database lets us use single segment reads rather than quorum reads except on recovery when the state is lost and has to be rebuilt.

The database may have multiple outstanding isolated transactions, which can complete (reach a finished and durable state) in a 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. The logic for tracking partially completed transactions and undoing them is kept in the database engine, just as if it were writing to simple disks. However, upon restart, before the database is allowed to access the storage volume, the storage service does its own recovery which is focused not on user-level transactions, but on making sure that the database sees a uniform view of storage despite its distributed nature.

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). 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.

Completeness and durability are therefore different and a CPL can be thought of as delineating some limited form of storage system transaction that must be accepted in order. If the client has no use for such distinctions, it can simply mark every log record as a CPL. 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.

On recovery, the database talks to the storage service to establish the durable point of each PG and uses that to establish the VDL and then issue commands to truncate the log records above VDL.

4.2 Normal Operation

We now describe the "normal operation" of the database engine and focus in turn on writes, reads, commits, and replicas.

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. For instance, 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. At any given moment, there can be a large number of concurrent transactions active in the database, each generating their own redo log records. 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). This limit ensures that the database does not get too far ahead of the storage system and introduces back-pressure that can throttle the incoming writes if the storage or network cannot keep up.

Note that 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. The SCL is used by the storage nodes when they gossip with each other in order to find and exchange log records that they are missing.

4.2.2 Commits

In Aurora, 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

In Aurora, as with most databases, pages are served from the buffer cache and only result in a storage IO request if the page in question is not present in the cache.

If the buffer cache is full, the system finds a victim page to evict from the cache. In a traditional system, if the victim is a "dirty page" then it is flushed to disk before replacement. This is to ensure that a subsequent fetch of the page always results in the latest data. While the Aurora database does not write out pages on eviction (or anywhere else), it enforces a similar guarantee: a page in the buffer cache must always be 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 that: (a) all changes in the page have been hardened in the log, and (b) 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 the 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.

The actual concurrency control protocols are executed in the database engine exactly as though the database pages and undo segments are organized in local storage as with traditional MySQL.

4.2.4 Replicas

In Aurora, a single writer and up to 15 read replicas can all mount a single shared storage volume. As a result, read replicas add no additional costs in terms of consumed storage or disk write operations. To minimize lag, 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. Note that the replicas consume log records asynchronously from the perspective of the writer, which acknowledges user commits independent of the replica. The replica obeys the following two important rules while applying log records: (a) the only log records that will be applied are those whose LSN is less than or equal to the VDL, and (b) 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. In practice, each replica typically lags behind the writer by a short interval (20 ms or less).

4.3 Recovery

Most traditional databases use a recovery protocol such as ARIES [7] that depends on the presence of a write-ahead log (WAL) that can represent the precise contents of all committed transactions. These systems also periodically checkpoint the database to establish points of durability in a coarse-grained fashion by flushing dirty pages to disk and writing a checkpoint record to the log. On restart, any given page can either miss some committed data or contain uncommitted data. Therefore, on crash recovery the system processes the redo log records since the last checkpoint by using the log applicator to apply each log record to the relevant database page. This process brings the database pages to a consistent state at the point of failure after which the in-flight transactions during the crash can be rolled back by executing the relevant undo log records. Crash recovery can be an expensive operation. Reducing the checkpoint interval helps, but at the expense of interference with foreground transactions. No such tradeoff is required with Aurora.

A great simplifying principle of a traditional database is that the same redo log applicator is used in the forward processing path as well as on recovery where it operates synchronously and in the foreground while the database is offline. We rely on the same principle in Aurora as well, except that the redo log applicator is decoupled from the database and operates on storage nodes, in parallel, and all the time in the background. Once the database starts up it performs volume recovery in collaboration with the storage service and as a result, an Aurora database can recover very quickly (generally under 10 seconds) even if it crashed while processing over 100,000 write statements per second.

The database does need to reestablish its runtime state after a crash. In this case, it 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 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. 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. PUTTING IT ALL TOGETHER

In this section, we describe the building blocks of Aurora as shown with a bird's eye view in Figure 5.

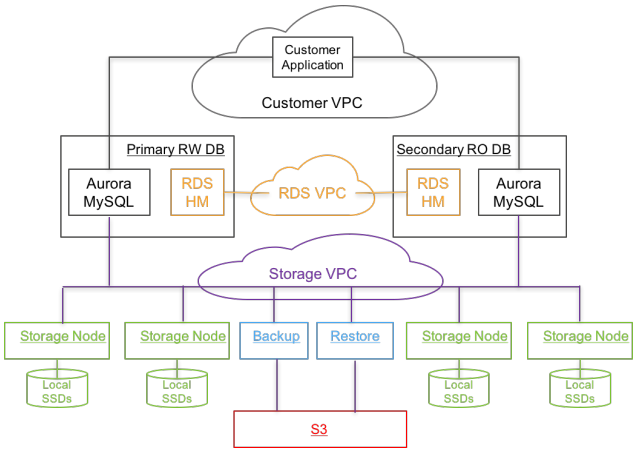

Figure 5: Aurora Architecture: A Bird's Eye View

The database engine is a fork of "community" MySQL/InnoDB and diverges primarily in how InnoDB reads and writes data to disk. In community InnoDB, a write operation results in data being modified in buffer pages, and the associated redo log records written to buffers of the WAL in LSN order. On transaction commit, the WAL protocol requires only that the redo log records of the transaction are durably written to disk. The actual modified buffer pages are also written to disk eventually through a double-write technique to avoid partial page writes. These page writes take place in the background, or during eviction from the cache, or while taking a checkpoint. In addition to the IO Subsystem, InnoDB also includes the transaction subsystem, the lock manager, a B+-Tree implementation and the associated notion of a "mini transaction" (MTR). An MTR is a construct only used inside InnoDB and models groups of operations that must be executed atomically (e.g., split/merge of B+-Tree pages).

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. The final log record of each MTR is tagged as a consistency point. Aurora supports exactly the same isolation levels that are supported by community MySQL in the writer (the standard ANSI levels and Snapshot Isolation or consistent reads). Aurora read replicas get continuous information on transaction starts and commits in the writer and use this information to support snapshot isolation for local transactions that are of course read-only. Note that concurrency control is implemented entirely in the database engine without impacting the storage service. The storage service presents a unified view of the underlying data that is logically identical to what you would get by writing the data to local storage in community InnoDB.

Aurora leverages Amazon Relational Database Service (RDS) for its control plane. RDS includes an agent on the database instance called the Host Manager (HM) that monitors a cluster's health and determines if it needs to fail over, or if an instance needs to be replaced. Each database instance is part of a cluster that consists of a single writer and zero or more read replicas. The instances of a cluster are in a single geographical region (e.g., us-east-1, us-west-1 etc.), are typically placed in different AZs, and connect to a storage fleet in the same region. For security, we isolate the communication between the database, applications and storage. In practice, each database instance can communicate on three Amazon Virtual Private Cloud (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 and is 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. The storage control plane uses the Amazon DynamoDB database service for persistent storage of cluster and storage volume configuration, volume metadata, and a detailed description of data backed up to S3. For orchestrating long-running operations, e.g. a database volume restore operation or a repair (re-replication) operation following a storage node failure, the storage control plane uses the Amazon Simple Workflow Service. Maintaining a high level of availability requires pro-active, automated, and early detection of real and potential problems, before end users are impacted. All critical aspects of storage operations are constantly monitored using metric collection services that raise alarms if key performance or availability metrics indicate a cause for concern.

6. PERFORMANCE RESULTS

In this section, we will share our experiences in running Aurora as a production service that was made "Generally Available" in July 2015. We begin with a summary of results running industry standard benchmarks and then present some performance results from our customers.

6.1 Results with Standard Benchmarks

Here we present results of different experiments that compare the performance of Aurora and MySQL using industry standard benchmarks such as SysBench and TPC-C variants. We ran MySQL on instances that are attached to an EBS volume with 30K provisioned IOPS. Except when stated otherwise, these are r3.8xlarge EC2 instances with 32 vCPUs and 244GB of memory

and features the Intel Xeon E5-2670 v2 (Ivy Bridge) processors. The buffer cache on the r3.8xlarge is set to 170GB.

6.1.1 Scaling with instance sizes

In this experiment, we report that throughput in Aurora can scale linearly with instance sizes, and with the highest instance size can be 5x that of MySQL 5.6 and MySQL 5.7. Note that Aurora is currently based on the MySQL 5.6 code base. We ran the SysBench read-only and write-only benchmarks for a 1GB data set (250 tables) on 5 EC2 instances of the r3 family (large, xlarge, 2xlarge, 4xlarge, 8xlarge). Each instance size has exactly half the vCPUs and memory of the immediately larger instance.

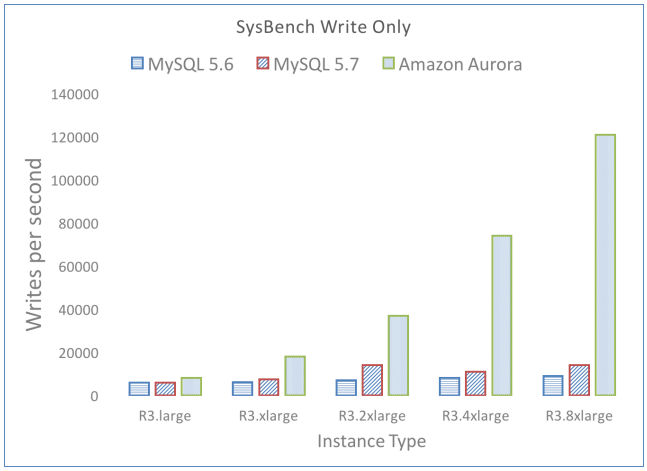

Figure 7: Aurora scales linearly for write-only workload

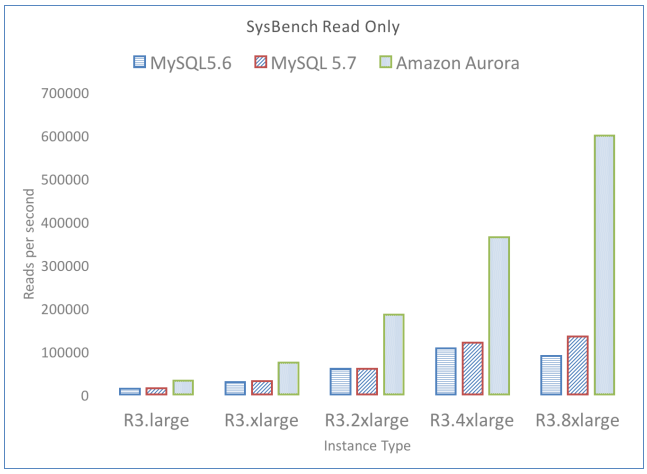

Figure 6: Aurora scales linearly for read-only workload

The results are shown in Figure 7 and Figure 6, and measure the performance in terms of write and read statements per second respectively. Aurora's performance doubles for each higher instance size and for the r3.8xlarge achieves 121,000 writes/sec and 600,000 reads/sec which is 5x that of MySQL 5.7 which tops out at 20,000 reads/sec and 125,000 writes/sec.

6.1.2 Throughput with varying data sizes

In this experiment, we report that throughput in Aurora significantly exceeds that of MySQL even with larger data sizes including workloads with out-of-cache working sets. Table 2 shows that for the SysBench write-only workload, Aurora can be up to 67x faster than MySQL with a database size of 100GB. Even for a database size of 1TB with an out-of-cache workload, Aurora is still 34x faster than MySQL.

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

DB Size	Amazon Aurora	MySQL
1 GB	107,000	8,400
10 GB	107,000	2,400
100 GB	101,000	1,500
1 TB	41,000	1,200

6.1.3 Scaling with user connections

In this experiment, we report that throughput in Aurora can scale with the number of client connections. Table 3 shows the results of running the SysBench OLTP benchmark in terms of writes/sec as the number of connections grows from 50 to 500 to 5000. While Aurora scales from 40,000 writes/sec to 110,000 writes/sec, the throughput in MySQL peaks at around 500 connections and then drops sharply as the number of connections grows to 5000.

Table 3: SysBench OLTP (writes/sec)

Connections	Amazon Aurora	MySQL
50	40,000	10,000
500	71,000	21,000
5,000	110,000	13,000

6.1.4 Scaling with Replicas

In this experiment, we report that the lag in an Aurora read replica is significantly lower than that of a MySQL replica even with more intense workloads. Table 4 shows that as the workload varies from 1,000 to 10,000 writes/second, the replica lag in Aurora grows from 2.62 milliseconds to 5.38 milliseconds. In contrast, the replica lag in MySQL grows from under a second to 300 seconds. At 10,000 writes/second Aurora has a replica lag that is several orders of magnitude smaller than that of MySQL. Replica lag is measured in terms of the time it takes for a committed transaction to be visible in the replica.

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

Writes/sec	Amazon Aurora	MySQL
1,000	2.62	< 1000
2,000	3.42	1000
5,000	3.94	60,000
10,000	5.38	300,000

6.1.5 Throughput with hot row contention

In this experiment, we report that Aurora performs very well relative to MySQL on workloads with hot row contention, such as those based on the TPC-C benchmark. We ran the Percona TPC-C variant [37] against Amazon Aurora and MySQL 5.6 and 5.7 on an r3.8xlarge where MySQL uses an EBS volume with 30K provisioned IOPS. Table 5 shows that Aurora can sustain between 2.3x to 16.3x the throughput of MySQL 5.7 as the workload varies from 500 connections and a 10GB data size to 5000 connections and a 100GB data size.

Table 5: Percona TPC-C Variant (tpmC)

Connections/Size/ Warehouses	Amazon Aurora	MySQL 5.6	MySQL 5.7
500/10GB/100	73,955	6,093	25,289
5000/10GB/100	42,181	1,671	2,592
500/100GB/1000	70,663	3,231	11,868
5000/100GB/1000	30,221	5,575	13,005

6.2 Results with Real Customer Workloads

In this section, we share results reported by some of our customers who migrated production workloads from MySQL to Aurora.

6.2.1 Application response time with Aurora

An internet gaming company migrated their production service from MySQL to Aurora on an r3.4xlarge instance. The average response time that their web transactions experienced prior to the migration was 15 ms. In contrast, after the migration the average response time 5.5 ms, a 3x improvement as shown in Figure 8.

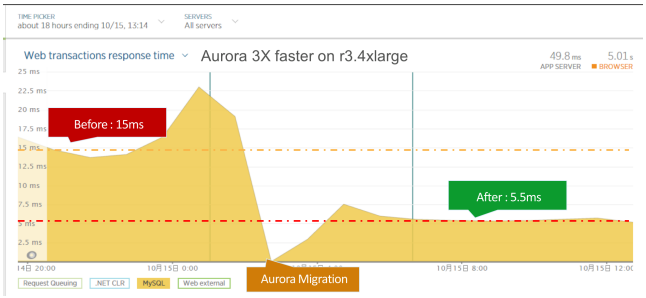

Figure 8: Web application response time

6.2.2 Statement Latencies with Aurora

An education technology company whose service helps schools manage student laptops migrated their production workload from MySQL to Aurora. The median (P50) and 95th percentile (P99) latencies for select and per-record insert operations before and after the migration (at 14:00 hours) are shown in Figure 9 and Figure 10.

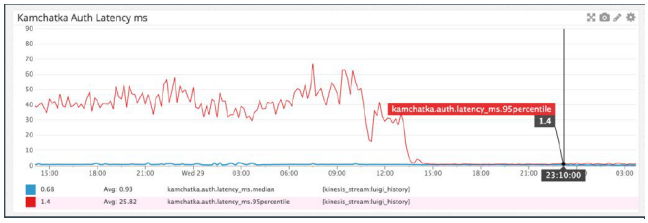

Figure 9: SELECT latency (P50 vs P95)

Before the migration, the P95 latencies ranged between 40ms to 80ms and were much worse than the P50 latencies of about 1ms. The application was experiencing the kinds of poor outlier performance that we described earlier in this paper. After the migration, however, the P95 latencies for both operations improved dramatically and approximated the P50 latencies.

Figure 10: INSERT per-record latency (P50 vs P95)

6.2.3 Replica Lag with Multiple Replicas

MySQL replicas often lag significantly behind their writers and can "can cause strange bugs" as reported by Weiner at Pinterest [40]. For the education technology company described earlier, the replica lag often spiked to 12 minutes and impacted application correctness and so the replica was only useful as a stand by. In contrast, after migrating to Aurora, the maximum replica lag across 4 replicas never exceeded 20ms as shown in Figure 11. The improved replica lag provided by Aurora let the company divert a significant portion of their application load to the replicas saving costs and increasing availability.

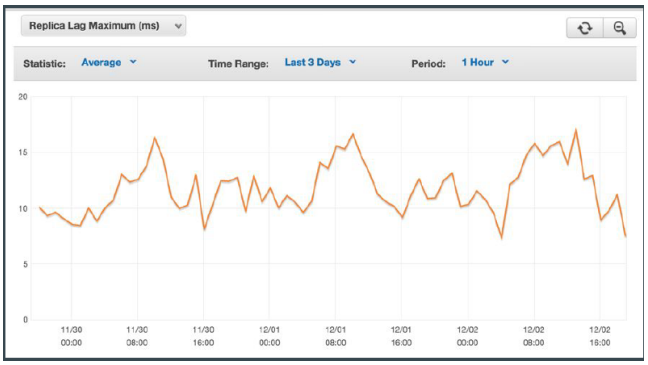

Figure 11: Maximum Replica Lag (averaged hourly)

7. LESSONS LEARNED

We have now seen a large variety of applications run by customers ranging from small internet companies all the way to highly sophisticated organizations operating large numbers of Aurora clusters. While many of their use cases are standard, we focus on scenarios and expectations that are common in the cloud and are leading us to new directions.

7.1 Multi-tenancy and database consolidation

Many of our customers operate Software-as-a-Service (SaaS) businesses, either exclusively or with some residual on-premise customers they are trying to move to their SaaS model. We find that these customers often rely on an application they cannot easily change. Therefore, they typically consolidate their different customers on a single instance by using a schema/database as a unit of tenancy. This idiom reduces costs: they avoid paying for a dedicated instance per customer when it is unlikely that all of their customers active at once. For instance, some of our SaaS customers report ha ving more than 50,000 customers of their own.

This model is markedly different from well-known multi-tenant applications like Salesforce.com [14] which use a multi-tenant data model and pack the data of multiple customers into unified tables of a single schema with tenancy identified on a per-row basis. As a result, we see many customers with consolidated databases containing a large number of tables. Production instances of over 150,000 tables for small database are quite common. This puts pressure on components that manage metadata like the dictionary cache. More importantly, such customers need (a) to sustain a high level of throughput and many concurrent user connections, (b) a model where data is only provisioned and paid for as it is used since it is hard to anticipate in advance how much storage space is needed, and (c) reduced jitter so that spikes for a single tenant have minimal impact on other tenants. Aurora supports these attributes and fits such SaaS applications very well.

7.2 Highly concurrent auto-scaling workloads

Internet workloads often need to deal with spikes in traffic based on sudden unexpected events. One of our major customers had a special appearance in a highly popular nationally televised show and experienced one such spike that greatly surpassed their normal peak throughput without stressing the database. To support such spikes, it is important for a database to handle many concurrent connections. This approach is feasible in Aurora since the underlying storage system scales so well. We have several customers that run at over 8000 connections per second.

7.3 Schema evolution

Modern web application frameworks such as Ruby on Rails deeply integrate object-relational mapping tools. As a result, it is easy for application developers to make many schema changes to their database making it challenging for DBAs to manage how the schema evolves. In Rails applications, these are called "DB Migrations" and we have heard first-hand accounts of DBAs that have to either deal with a "few dozen migrations a week", or put in place hedging strategies to ensure that future migrations take place without pain. The situation is exacerbated with MySQL offering liberal schema evolution semantics and implementing most changes using a full table copy. Since frequent DDL is a pragmatic reality, we have implemented an efficient online DDL implementation that (a) versions schemas on a per-page basis and decodes individual pages on demand using their schema history, and (b) lazily upgrades individual pages to the latest schema using a modify-on-write primitive.

7.4 Availability and Software Upgrades

Our customers have demanding expectations of cloud-native databases that can conflict with how we operate the fleet and how often we patch servers. Since our customers use Aurora primarily as an OLTP service backing production applications, any disruption can be traumatic. As a result, many of our customers have a very low tolerance to our updates of database software, even if this amounts to a planned downtime of 30 seconds every 6 weeks or so. Therefore, we recently released a new Zero-Downtime Patch (ZDP) feature that allows us to patch a customer while in-flight database connections are unaffected.

As shown in Figure 12, ZDP works by looking for an instant where there are no active transactions, and in that instant spooling the application state to local ephemeral storage, patching the engine and then reloading the application state. In the process, user sessions remain active and oblivious that the engine changed under the covers.

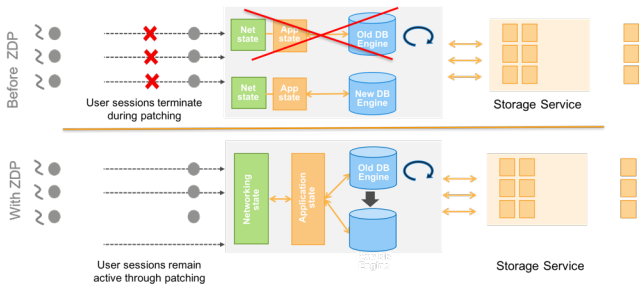

Figure 12: Zero-Downtime Patching

8. RELATED WORK

In this section, we discuss other contributions and how they relate to the approaches taken in Aurora.

Decoupling storage from compute. Although traditional systems have usually been built as monolithic daemons [27], there has been recent work on databases that decompose the kernel into different components. For instance, Deuteronomy [10] is one such system that separates a Transaction Component (TC) that provides concurrency control and recovery from a Data Component (DC) that provides access methods on top of LLAMA [34], a latch-free log-structured cache and storage manager. Sinfonia [39] and Hyder [38] are systems that abstract transactional access methods over a scale out service and database systems can be implemented using these abstractions. The Yesquel [36] system implements a multi-version distributed balanced tree and separates concurrency control from the query processor. Aurora decouples storage at a level lower than that of Deuteronomy, Hyder, Sinfonia, and Yesquel. In Aurora, query processing, transactions, concurrency, buffer cache, and access methods are decoupled from logging, storage, and recovery that are implemented as a scale out service.

Distributed Systems. The trade-offs between correctness and availability in the face of partitions have long been known with the major result that one-copy serializability is not possible in the face of network partitions [15]. More recently Brewer's CAP Theorem as proved in [16] stated that a highly available system cannot provide "strong" consistency guarantees in the presence of network partitions. These results and our experience with cloud-scale complex and correlated failures motivated our consistency goals even in the presence of partitions caused by an AZ failure.

Bailis et al [12] study the problem of providing Highly Available Transactions (HATs) that neither suffer unavailability during partitions nor incur high network latency. They show that Serializability, Snapshot Isolation and Repeatable Read isolation are not HAT-compliant, while most other isolation levels are achievable with high availability. Aurora provides all these isolation levels by making a simplifying assumption that at any time there is only a single writer generating log updates with LSNs allocated from a single ordered domain.

Google's Spanner [24] provides externally consistent [25] reads and writes, and globally-consistent reads across the database at a timestamp. These features enable Spanner to support consistent backups, consistent distributed query processing [26], and atomic schema updates, all at global scale, and even in the presence of ongoing transactions. As explained by Bailis [12], Spanner is highly specialized for Google's read-heavy workload and relies on two-phase commit and two-phase locking for read/write transactions.

Concurrency Control. Weaker consistency (PACELC [17]) and isolation models [18][20] are well known in distributed databases and have led to optimistic replication techniques [19] as well as eventually consistent systems [21][22][23]. Other approaches in centralized systems range from classic pessimistic schemes based on locking [28], optimistic schemes like multi-versioned concurrency control in Hekaton [29], sharded approaches such as VoltDB [30] and Timestamp ordering in HyPer [31][32] and Deuteronomy. Aurora's storage service provides the database engine the abstraction of a local disk that is durably persisted, and allows the engine to determine isolation and concurrency control.

Log-structured storage. Log-structured storage systems were introduced by LFS [33] in 1992. More recently Deuteronomy and the associated work in LLAMA [34] and Bw-Tree [35] use log-structured techniques in multiple ways across the storage engine stack and, like Aurora, reduce write amplification by writing deltas instead of whole pages. Both Deuteronomy and Aurora implement pure redo logging, and keep track of the highest stable LSN for acknowledging commits.

Recovery. While traditional databases rely on a recovery protocol based on ARIES [5], some recent systems have chosen other paths for performance. For example, Hekaton and VoltDB rebuild their in-memory state after a crash using some form of an update log. Systems like Sinfonia [39] avoid recovery by using techniques like process pairs and state machine replication. Graefe [41] describes a system with per-page log record chains that enables on-demand page-by-page redo that can make recovery fast. Like Aurora, Deuteronomy does not require redo recovery. This is because Deuteronomy delays transactions so that only committed updates are posted to durable storage. As a result, unlike Aurora, the size of transactions can be constrained in Deuteronomy.

9. CONCLUSION

We designed Aurora as a high throughput OLTP database that compromises neither availability nor durability in a cloud-scale environment. The big idea was to move away from the monolithic architecture of traditional databases and decouple storage from compute. In particular, we moved the lower quarter of the database kernel to an independent scalable and distributed service that managed logging and storage. With all I/Os written over the network, our fundamental constraint is now the network. As a result we need to focus on techniques that relieve the network and improve throughput. We rely on quorum models that can handle the complex and correlated failures that occur in large-scale cloud environments and avoid outlier performance penalties, log processing to reduce the aggregate I/O burden, and asynchronous consensus to eliminate chatty and expensive multi-phase synchronization protocols, offline crash recovery, and checkpointing in distributed storage. Our approach has led to a simplified architecture with reduced complexity that is easy to scale as well as a foundation for future advances.

10. ACKNOWLEDGMENTS

We thank the entire Aurora development team for their efforts on the project including our current members as well as our distinguished alumni (James Corey, Sam McKelvie, Yan Leshinsky, Lon Lundgren, Pradeep Madhavarapu, and Stefano Stefani). We are particularly grateful to our customers who operate production workloads using our service and have been generous in sharing their experiences and expectations with us. We also thank the shepherds for their invaluable comments in shaping this paper.

11. REFERENCES

[1] B. Calder, J. Wang, et al. Windows Azure storage: A highly available cloud storage service with strong consistency. In *SOSP 2011*.

[2] O. Khan, R. Burns, J. Plank, W. Pierce, and C. Huang. Rethinking erasure codes for cloud file systems: Minimizing I/O for recovery and degraded reads. In *FAST 2012*.

[3] P.A. Bernstein, V. Hadzilacos, and N. Goodman. Concurrency control and recovery in database systems, Chapter 7, Addison Wesley Publishing Company, ISBN 0-201-10715-5, 1997.

[4] C. Mohan, B. Lindsay, and R. Obermarck. Transaction management in the R* distributed database management system". *ACM TODS,* 11(4):378-396, 1986.

[5] C. Mohan and B. Lindsay. Efficient commit protocols for the tree of processes model of distributed transactions. *ACM SIGOPS Operating Systems Review*, 19(2):40-52, 1985.

[6] D.K. Gifford. Weighted voting for replicated data. In *SOSP 1979*.

[7] C. Mohan, D.L. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM TODS*, 17 (1): 94–162, 1992.

[8] R. van Renesse and F. Schneider. Chain replication for supporting high throughput and availability. In *OSDI 2004*.

[9] A. Kopytov. Sysbench Manual. Available at http://imysql.com/wp-content/uploads/2014/10/sysbench-manual.pdf

[10] J. Levandoski, D. Lomet, S. Sengupta, R. Stutsman, and R. Wang. High performance transactions in deuteronomy. In *CIDR 2015*.

[11] P. Bailis, A. Fekete, A. Ghodsi, J.M. Hellerstein, and I. Stoica. Scalable atomic visibility with RAMP Transactions. In *SIGMOD 2014*.

[12] P. Bailis, A. Davidson, A. Fekete, A. Ghodsi, J.M. Hellerstein, and I. Stoica. Highly available transactions: virtues and limitations. In *VLDB* 2014.

[13] R. Taft, E. Mansour, M. Serafini, J. Duggan, A.J. Elmore, A. Aboulnaga, A. Pavlo, and M. Stonebraker. E-Store: fine-grained elastic partitioning for distributed transaction processing systems. In *VLDB 2015*.

[14] R. Woollen. The internal design of salesforce.com's multi-tenant architecture. In *SoCC 2010*.

[15] S. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in partitioned networks. *ACM CSUR*, 17(3):341–370, 1985.

[16] S. Gilbert and N. Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[17] D.J. Abadi. Consistency tradeoffs in modern distributed database system design: CAP is only part of the story. *IEEE Computer*, 45(2), 2012.

[18] A. Adya. Weak consistency: a generalized theory and optimistic implementations for distributed transactions. PhD Thesis, MIT, 1999.

[19] Y. Saito and M. Shapiro. Optimistic replication. *ACM Comput. Surv.*, 37(1), Mar. 2005.

[20] H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil. A critique of ANSI SQL isolation levels. In *SIGMOD 1995*.

[21] P. Bailis and A. Ghodsi. Eventual consistency today: limitations, extensions, and beyond. *ACM Queue*, 11(3), March 2013.

[22] P. Bernstein and S. Das. Rethinking eventual consistency. In *SIGMOD*, 2013.

[23] B. Cooper et al. PNUTS: Yahoo!'s hosted data serving platform. In *VLDB 2008*.

[24] J. C. Corbett, J. Dean, et al. Spanner: Google's globally-distributed database. In *OSDI 2012*.

[25] David K. Gifford. Information Storage in a Decentralized Computer System. *Tech. rep. CSL-81-8*. PhD dissertation. Xerox PARC, July 1982.

[26] Jeffrey Dean and Sanjay Ghemawat. MapReduce: a flexible data processing tool". *CACM* 53 (1):72-77, 2010.

[27] J. M. Hellerstein, M. Stonebraker, and J. R. Hamilton. Architecture of a database system. *Foundations and Trends in Databases*. 1(2) pp. 141-259, 2007.

[28] J. Gray, R. A. Lorie, G. R. Putzolu, I. L. Traiger. Granularity of locks in a shared data base. In *VLDB 1975*.

[29] P-A Larson, et al. High-Performance Concurrency control mechanisms for main-memory databases. *PVLDB*, 5(4): 298-309, 2011.

[30] M. Stonebraker and A. Weisberg. The VoltDB main memory DBMS. *IEEE Data Eng. Bull.*, 36(2): 21-27, 2013.

[31] V. Leis, A. Kemper, et al. Exploiting hardware transactional memory in main-memory databases. In *ICDE 2014*.

[32] H. Mühe, S. Wolf, A. Kemper, and T. Neumann: An evaluation of strict timestamp ordering concurrency control for main-memory database systems. *In IMDM 2013*.

[33] M. Rosenblum and J. Ousterhout. The design and implementation of a log-structured file system. *ACM TOCS* 10(1): 26–52, 1992.

[34] J. Levandoski, D. Lomet, S. Sengupta. LLAMA: A cache/storage subsystem for modern hardware. PVLDB 6(10): 877-888, 2013.

[35] J. Levandoski, D. Lomet, and S. Sengupta. The Bw-Tree: A B-tree for new hardware platforms. In *ICDE 2013*.

[36] M. Aguilera, J. Leners, and M. Walfish. Yesquel: scalable SQL storage for web applications. In *SOSP 2015*.

[37] Percona Lab. TPC-C Benchmark over MySQL. Available at https://github.com/Percona-Lab/tpcc-mysql

[38] P. Bernstein, C. Reid, and S. Das. Hyder – A transactional record manager for shared flash. In *CIDR 2011*.

[39] M. Aguilera, A. Merchant, M. Shah, A. Veitch, and C. Karamanolis. Sinfonia: A new paradigm for building scalable distributed systems. ACM Trans. Comput. Syst. 27(3): 2009.

[40] M. Weiner. Sharding Pinterest: How we scaled our MySQL fleet. Pinterest Engineering Blog. Available at: https://engineering.pinterest.com/blog/sharding-pinterest-how-we-scaled-our-mysql-fleet

[41] G. Graefe. Instant recovery for data center savings. *ACM SIGMOD Record*. 44(2):29-34, 2015.

[42] J. Dean and L. Barroso. The tail at scale. CACM 56(2):74-80, 2013.