Amazon Aurora

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

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.

Amazon Aurora是一种关系数据库服务，用于作为Amazon Web服务（AWS）的一部分提供的OLTP工作负载。在本文中，我们描述了Aurora的体系结构以及主导该体系结构的设计考虑。我们认为，高吞吐数据处理的核心约束已从计算和存储转移到网络。Aurora为关系数据库带来了一种新的体系结构来解决这一约束，最显著的是将redo处理推送到多租户扩展存储服务，该服务是专为Aurora构建的。我们描述了这样做不仅可以减少网络流量，还可以实现快速故障恢复、在不丢失数据的情况下切换到副本，以及容错、自愈存储。然后，我们描述了Aurora如何使用高效的异步方案在多个存储节点上实现持久状态共识，从而避免昂贵和频繁的恢复协议。最后，在将Aurora作为一项生产服务运营了18个月之后，我们分享了我们从客户那里学到的关于现代云应用程序对其数据库层的期望的经验教训。

Keywords

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

一、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.

IT工作负载越来越多地转移到公共云提供商。这一全行业转型的重要原因包括能够在灵活的按需基础上提供容量，并使用运营费用而不是资本费用模式来支付此容量。许多IT工作负载需要关系型OLTP数据库；为本地数据库提供同等或卓越的功能对于支持这种长期过渡至关重要。

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.

在现代分布式云服务中，通过将计算与存储分离[10]、[24]、[36]、[38]、[39]以及跨多个节点复制存储，弹性和可扩展性日益增强。这样做可以让我们处理如替换行为不正常或无法访问的主机、添加副本、从写入程序故障切换到副本、向上或向下扩展数据库实例的大小等操作。

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.

传统数据库系统面临的I/O瓶颈在此环境中发生了变化。由于I/O可以分布在多租户群中的多个节点和多个磁盘上，因此单个磁盘和节点不再是热的。相反，瓶颈转移到请求I/O的数据库层和执行这些I/O操作的存储层之间的网络。除了每秒数据包数（PPS）和带宽的基本瓶颈之外，还有流量的放大，因为性能良好的数据库将并行向存储组发出写操作。异常存储节点、磁盘或网络路径的性能可能会主导响应时间。

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.

虽然数据库中的大多数操作可能相互重叠，但有几种情况需要同步操作。这会导致暂停和上下文切换。其中一种情况是由于数据库缓冲区缓存中的未命中而导致的磁盘读取。读取线程在其读取完成之前无法继续。缓存未命中还可能导致额外的损耗，即逐出并刷新脏缓存页以容纳新页。后台处理（如检查点和脏页写入）可以减少这种损耗的发生，但也可能导致暂停、上下文切换和资源争用。

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.

事务提交是另一个干扰源；提交一个事务的停顿可能会阻止其他事务的进行。在云级分布式系统中，使用多阶段同步协议（如2阶段提交（2PC）[3][4][5]）处理提交是一项挑战。这些协议不能容忍故障，并且大规模分布式系统具有持续的硬故障和软故障"背景噪声"。由于大规模系统分布在多个数据中心，因此它们的延迟也很高。

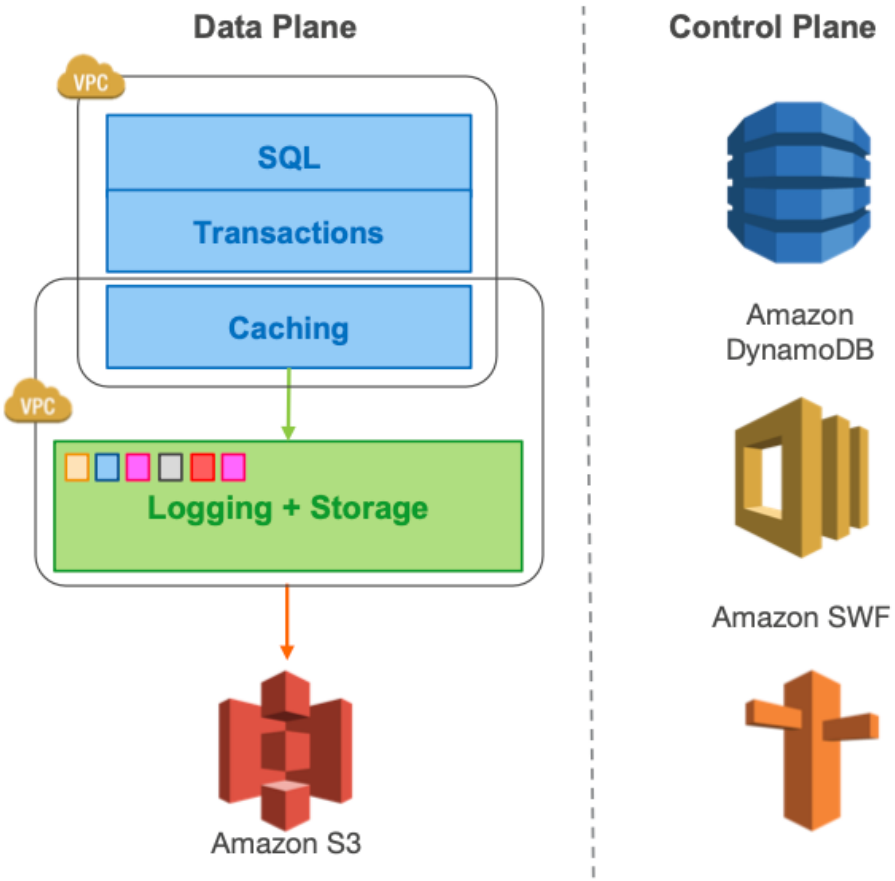

Figure 1: Move logging and storage off the database engine

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.

在本文中，我们介绍了Amazon Aurora，这是一种新的数据库服务，通过在高度分布式的云环境中更积极地利用redo日志来解决上述问题。我们使用了一种新的面向服务的体系结构（见图1），该体系结构具有多租户扩展存储服务，该服务抽象了虚拟化分段redo日志，并松散地耦合到一组数据库实例。虽然每个实例仍然包含传统内核的大部分组件（查询处理器、事务、锁定、缓冲区缓存、访问方法和撤消管理），但有几个功能（redo日志记录、持久存储、崩溃恢复和备份/恢复）已卸载到存储服务中。

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.

与传统方法相比，我们的架构有三个显著优势。首先，通过将存储构建为跨多个数据中心的独立容错和自愈服务，我们可以在网络层或存储层保护数据库免受性能变化和暂时或永久故障的影响。我们观察到，耐久性故障可以建模为长期可用性事件，可用性事件可以建模为长时间性能变化——设计良好的系统可以统一处理这些事件[42]。其次，通过仅将redo日志记录写入存储，我们能够将网络IOPS降低一个数量级。一旦我们消除了这个瓶颈，我们就能够积极地优化许多其他争用点，与我们开始使用的基本MySQL代码相比，获得了显著的吞吐量改进。第三，我们将一些最复杂和关键的功能（备份和重做恢复）从数据库引擎中的一次性昂贵操作转移到在大型分布式车队中摊销的连续异步操作。这可以实现近乎即时的崩溃恢复，而无需检查点，而且备份成本低廉，不会干扰前台处理。

In this paper, we describe three contributions:

在本文中，我们描述了三个贡献：

1. How to reason about durability at cloud scale and how to design quorum systems that are resilient to correlated failures. (Section 2).

   如何在云范围内对耐用性进行推理，以及如何设计能够抵御相关故障的quorum系统。（第2节）

2. How to leverage smart storage by offloading the lower quarter of a traditional database to this tier. (Section 3).

如何通过将传统数据库的下四分之一卸载到此层来利用智能存储。（第3节）。

3. How to eliminate multi-phase synchronization, crash recovery and checkpointing in distributed storage(Section 4).

如何消除分布式存储中的多阶段同步、崩溃恢复和检查点（第4节）

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.

然后，我们将在第5节中展示如何将这三个想法结合在一起设计Aurora的整体架构，然后在第6节中回顾我们的性能结果，并在第7节中总结经验教训。最后，我们简要回顾了第8节中的相关工作，并在9节中提出了结论

二、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.

如果数据库系统不做其他事情，它必须满足数据一旦写入就可以读取的契约。并非所有系统都这样做。在本节中，我们将讨论quorum模型背后的基本原理，为什么我们要对存储进行细分，以及二者结合起来如何不仅提供耐用性、可用性和抖动减少，而且帮助我们解决大规模管理存储机群的运营问题。

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.

实例生存期与存储生存期没有很好的相关性。实例失败。客户关闭了它们。它们根据负载上下调整大小。由于这些原因，它有助于将存储层与计算层解耦。

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.

一旦这样做，这些存储节点和磁盘也可能发生故障。因此，必须以某种形式复制它们，以提供故障恢复能力。在大规模云环境中，节点、磁盘和网络路径故障的底层干扰持续较低。每个故障可能具有不

同的持续时间和不同的爆炸半径。例如，一个节点可能暂时缺乏网络可用性，重新启动时出现临时停机，或者磁盘、节点、机架、叶子或脊椎网络交换机，甚至数据中心出现永久故障。

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 Vr votes or a write quorum of Vw votes. To achieve consistency, the quorums must obey two rules. First, each read must be aware of the most recent write, formulated as Vr + Vw > 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 Vw > V/2.

在复制系统中容忍故障的一种方法是使用[6]中描述的基于仲裁的投票协议。如果为复制数据项的V个副本中的每个副本分配了投票，则读或写操作必须分别获得Vr投票的读法定人数或Vw投票的写法定人数。为了达到一致性，法定人必须遵守两条规则。首先，每次读取必须知道最近的写入，公式为Vr+Vw>V。此规则确保用于读取的节点集与用于写入的节点集相交，并且读取仲裁至少包含一个具有最新版本的位置。其次，每次写入必须知道最近的写入，以避免冲突写入，公式为Vw>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 (Vw=2)andareadquorumof2/3(Vr=2).

容忍单个节点丢失的常见方法是将数据复制到（V=3）个节点，并依赖于2/3（Vw=2）的写入仲裁和2/3的读取仲裁（Vr=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.

我们认为2/3的数量是不够的。为了理解原因，让我们首先了解AWS中可用区（AZ）的概念。AZ是一个区域的子集，该区域通过低延迟链路连接到该区域中的其他AZ，但对于大多数故障（包括电源、网络、软件部署、洪水等）是隔离的。跨AZ分发数据副本可确保大规模的典型故障模式仅影响一个数据副本。这意味着可以简单地将三个副本中的每一个放置在不同的AZ中，并且除了较小的单个故障外，还可以容忍大规模事件。

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.

然而，在大型存储机队中，故障的底层干扰意味着，在任何给定时刻，某些磁盘或节点可能已发生故障并正在修复。这些故障可能独立地分布在AZ A、B和C中的每个节点上。但是，由于火灾、屋顶故障、洪水等，AZ C的故障将破坏AZ A或AZ B中同时发生故障的任何副本的仲裁。此时，在2/3读取仲裁模型中，我们将丢失两个副本，并且无法确定第三个副本是否是最新的。换句话说，虽然每个AZ中副本的单个故障是不相关的，但AZ的故障是该AZ中所有磁盘和节点的相关故障。Quorum需要容忍AZ故障以及同时发生的底层干扰故障。

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 (Vw = 4), and a read quorum of 3/6 (Vr = 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.

在Aurora中，我们选择了一个设计点，允许（a）在不丢失数据的情况下丢失整个AZ和一个额外节点（AZ+1），以及（b）在不影响数据写入能力的情况下损失整个AZ。我们通过在3个AZ上以6种方式复制每个数据项来实现这一点，每个AZ中有每个数据项的2个副本。我们使用的法定人数模型有6票（V=6），写入法定人数为4/6（Vw=4），读取法定人数为3/6（Vr=3）。使用这种模型，我们可以（a）在不丢失读可用性的情况下丢失一个AZ和一个附加节点（3个节点的故障），以及（b）丢失任何两个节点，包括一个AX故障并保持写可用性。确保读仲裁使我们能够通过添加其他副本副本来重建写仲裁。

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.

让我们考虑AZ+1是否提供足够的耐久性的问题。为了在该模型中提供足够的耐久性，必须确保在修复其中一个故障（平均修复时间–MTTR）所需的时间内，不相关故障（平均失效时间–MTTF）的双重故障概率足够低。如果双重故障的概率足够高，我们可能会在AZ故障上看到这些，从而打破定额。在一定程度上，很难降低独立故障的MTTF概率。相反，我们专注于降低MTTR，以缩小双故障的漏洞窗口。我们通过将数据库卷划分为固定大小的小段来实现这一点，目前大小为10GB。这些都以6种方式复制到保护组（PG）中，因此每个PG由六个10GB段组成，跨三个AZ组织，每个AZ中有两个段。存储卷是一组级联的PG，使用大量存储节点进行物理实现，这些节点被配置为虚拟主机，并使用Amazon弹性计算云（EC2）连接SSD。构成卷的PG随着卷的增长而分配。我们目前支持的容量可以在不重复的基础上增长到64 TB。

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.

分段现在是我们的独立背景噪声故障和修复单元。作为我们服务的一部分，我们监控并自动修复故障。10Gbps网络链路上的10GB网段可以在10秒内修复。我们需要在同一个10秒窗口中看到两个这样的故障，再加上一个不包含这两个独立故障的AZ故障，以失去法定人数。根据我们观察到的故障率，即使对于我们为客户管理的数据库数量来说，这也是不太可能的

2.3 Operational Advantages of Resilience（弹性的运行优势）

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.

一旦设计出了一个能够自然抵御长期故障的系统，它自然也能抵御较短的故障。可以处理AZ长期丢失的存储系统也可以处理因电源事件或需要回滚的错误软件部署而导致的短暂停机。能够处理仲裁成员数秒可用性损失的服务器可以处理存储节点上的短暂网络拥塞或负载

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.

由于我们的系统具有很高的故障容忍度，我们可以利用这一点进行导致段不可用的维护操作。例如，热管理非常简单。我们可以将一个热磁盘或节点上的一个段标记为坏的，通过迁移到集群中其他较冷的节点，仲裁将被快速修复。操作系统和安全修补是该存储节点在修补时发生的短暂不可用事件。甚至我们的存储车队的软件升级也以这种方式进行管理。我们一次执行一个AZ，并确保同时修补的PG成员不超过一个。这使我们能够在存储服务中使用敏捷方法和快速部署。

三、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.

在本节中，我们将解释为什么在分段复制存储系统上使用传统数据库（如第2节所述）会在网络IOs和同步暂停方面带来无法承受的性能负担。然后，我们解释了将日志处理卸载到存储服务的方法，并通过实验演示了我们的方法如何显著减少网络IOs。最后，我们描述了存储服务中用于最小化同步暂停和不必要写入的各种技术。

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.

我们对存储卷进行分段，并使用4/6写入仲裁以6种方式复制每个分段的模型使我们具有很高的弹性。不幸的是，对于MySQL这样的传统数据库，这种模型会导致无法维持的性能，因为它会为每个应用程序写入生成许多不同的实际I/O。复制放大了高I/O容量，造成了沉重的每秒数据包（PPS）负担。此外，I/O会导致管道停滞并延长延迟的同步点。虽然链复制[8]及其替代方案可以降低网络成本，但它们仍然存在同步暂停和附加延迟问题。

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.

让我们来看看在传统数据库/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.

实际上，还必须写入其他数据。例如，考虑一个同步镜像MySQL配置，该配置可实现跨数据中心的高可用性，并在主备配置中运行，如图2所示。AZ1中有一个活动MySQL实例，在Amazon弹性块存储（EBS）上具有网络存储。AZ2中还有一个备用MySQL实例，在EBS上也有网络存储。使用软件镜像将对主EBS卷的写入与备用EBS卷同步。

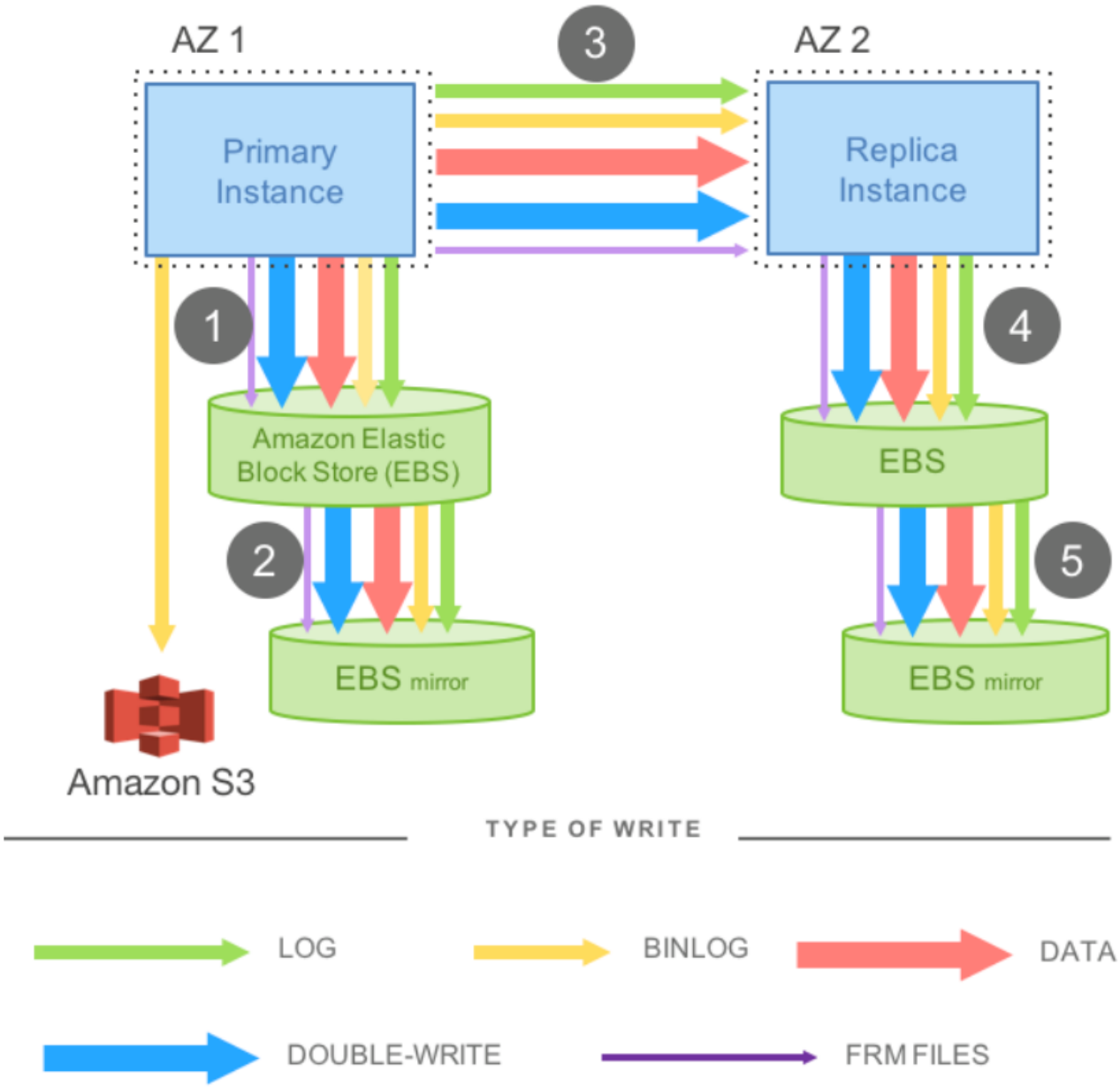

Figure 2: Network IO in mirrored MySQL

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.

图2显示了引擎需要写入的各种类型的数据：重做日志、归档到Amazon Simple Storage Service（S3）以支持时间点恢复的二进制（语句）日志、修改后的数据页、第二次临时写入数据页（双重写入）以防止页面撕裂，最后是元数据（FRM）文件。该图还显示了实际IO流的顺序，如下所示。在步骤1和2中，向EBS发出写操作，然后EBS将其发送到AZ本地镜像，并在两者都完成时接收到确认。接下来，在步骤3中，使用同步块级软件镜像将写入分阶段到备用实例。最后，在步骤4和5中，将写入操作写入备用EBS卷和相关镜像。

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.

上面描述的镜像MySQL模型是不受欢迎的，不仅因为数据是如何写入的，还因为数据是什么写入的。首先，步骤1、3和5是顺序和同步的。延迟是加性的，因为许多写入是连续的。抖动被放大，因为即使在异步写操作中，也必须等待最慢的操作，使系统受到异常值的影响。从分布式系统的角度来看，该模型可以被视为具有4/4写入仲裁，并且容易出现故障和异常性能。其次，OLTP应用程序导致的用户操作会导致许多不同类型的写入，通常以多种方式表示相同的信息——例如，写入双写缓冲区，以防止存储基础结构中的页被撕碎

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.

当传统数据库修改数据页时，它会生成一个重做日志记录，并调用日志应用程序，该应用程序将重做日志纪录应用于该页的内存中的前映像，以生成其后映像。事务提交要求写入日志，但数据页写入可能会延迟。

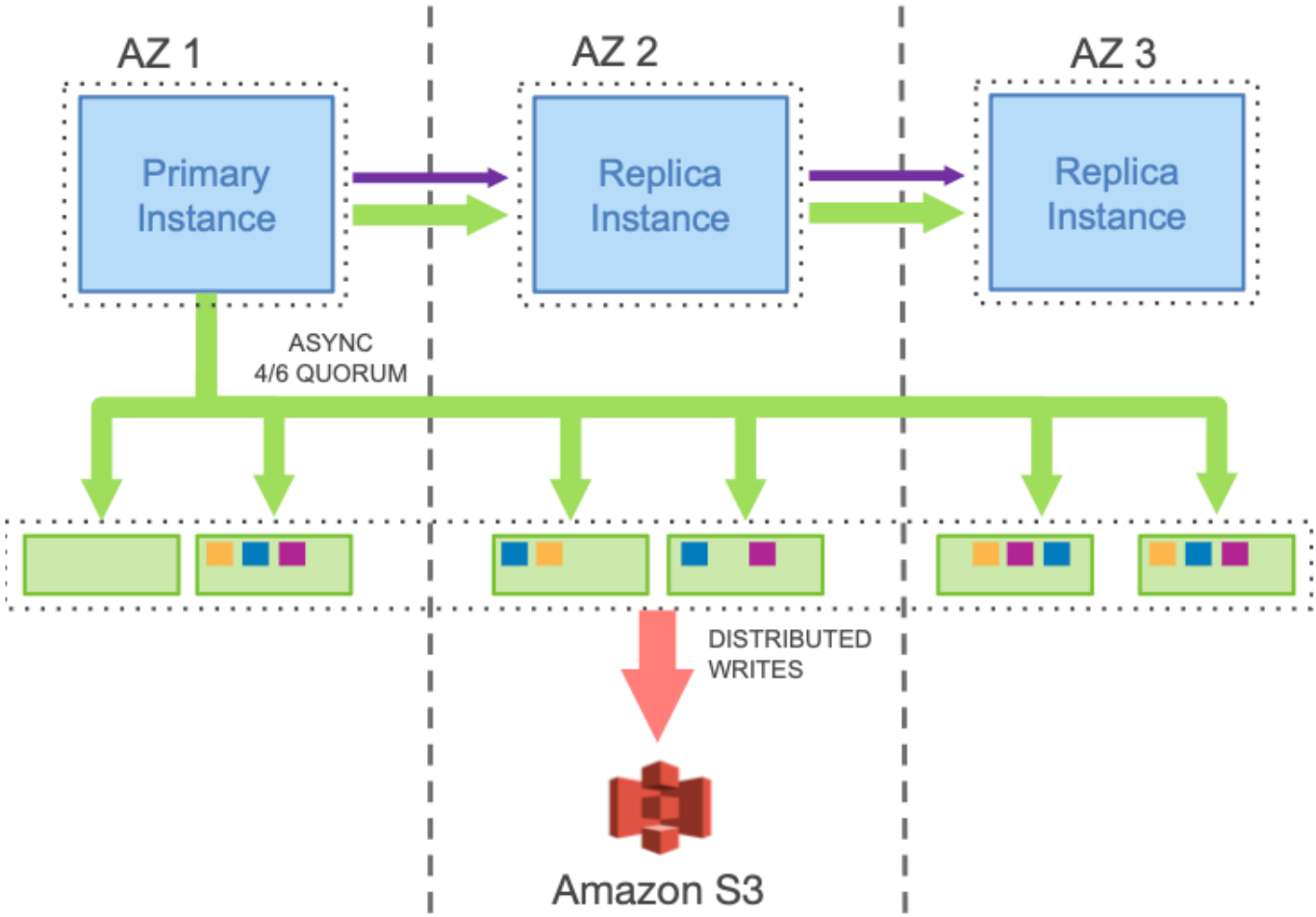

Figure 3: Network IO in Amazon Aurora

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.

在Aurora中，跨网络的唯一写入是重做日志记录。从未从数据库层写入任何页面，不是为了后台写入，不是为了检查点，也不是为了缓存逐出。相反，日志应用程序被推送到存储层，可以在后台或按需生成数据库页面。当然，从一开始就从完整的修改链生成每个页面的成本高得惊人。因此，我们不断在后台具体化数据库页面，以避免每次都根据需要从头开始重新生成它们。请注意，从正确性的角

度来看，后台具体化是完全可选的：就引擎而言，日志就是数据库，存储系统具体化的任何页面都只是日志应用程序的缓存。还请注意，与检查点不同，只有具有长修改链的页面需要重物化。检查点由整个重做日志链的长度控制。Aurora页面物化由给定页面的链长度控制。

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.

我们的方法极大地减少了网络负载，尽管放大了复制写入，并提供了性能和耐用性。存储服务可以以令人尴尬的并行方式扩展I/O，而不会影响数据库引擎的写入吞吐量。例如，图3显示了一个Aurora集群，其中一个主实例和多个副本实例部署在多个AZ上。在此模型中，主服务器仅将日志记录写入存储服务，并将这些日志记录以及元数据更新流式传输到副本实例。IO流基于公共目标（逻辑段，即PG）对完全排序的日志记录进行批处理，并将每个批传递给所有6个副本，其中该批被持久保存在磁盘上，数据库引擎等待来自6个副本中4个副本的确认，以满足写仲裁，并将所讨论的日志记录视为持久或加固的。复制副本使用重做日志记录将更改应用于其缓冲区缓存。

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.

为了测量网络I/O，我们使用SysBench[9]纯写工作负载对上述两种配置进行了测试，其中一种配置具有跨多个AZ的同步镜像MySQL配置，另一种配置为RDS Aurora（跨多个AZ的副本）。在这两种情况下，测试都针对运行在r3.8x2大型EC2实例上的数据库引擎运行了30分钟。

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.

我们的实验结果总结在表1中。在30分钟内，Aurora能够支持比镜像MySQL多35倍的事务。Aurora中的数据库节点上的每个事务的I/O数是镜像MySQL中的7.7倍，尽管Aurora将写操作放大了六倍，并且没有计算EBS中的链式复制，也没有计算MySQL中交叉AZ写操作。每个存储节点都会看到未扩展的写入，因为它只是六个副本中的一个，因此需要在这一层进行处理的I/O数量减少了46倍。我们通过向网络写入更少的数据而获得的节约让我们能够积极地复制数据以获得持久性和可用性，并并行发出请求以最小化抖动的影响。

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.

将处理转移到存储服务还可以通过最小化崩溃恢复时间来提高可用性，并消除后台进程（如检查点、后台数据页写入和备份）造成的抖动。

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.

让我们检查一下崩溃恢复。在传统数据库中，崩溃后，系统必须从最近的检查点开始并重放日志，以确保已应用所有持久的重做记录。在Aurora中，持久的重做记录应用程序发生在存储层，连续、异步并分布在整个机群中。如果数据页不是当前页，则对数据页的任何读取请求都可能需要应用一些重做记录。因此，崩溃恢复过程扩展到所有正常的前台处理。数据库启动时不需要任何东西。

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.

我们存储服务的一个核心设计原则是将前台写请求的延迟降至最低。我们将大部分存储处理移到后台。鉴于来自存储层的前台请求峰值与平均值之间的自然差异，我们有充足的时间在前台路径之外执行这些任务。我们还有机会用CPU换磁盘。例如，当存储节点忙于处理前台写请求时，不必对旧页面版本运行垃圾收集（GC），除非磁盘接近容量。在极光中，背景处理与前景处理呈负相关。这与传统数据库不同，在传统数据库中，页面的后台写入和检查点与系统上的前台负载呈正相关。如果我们在系统上建立了一个积压，我们将限制前台活动，以防止长队列的积累。由于在我们的系统中，段以高熵分布在各个存储节点上，因此一个存储节点上的节流很容易由我们的4/6仲裁写入处理，表现为一个缓慢的节点。

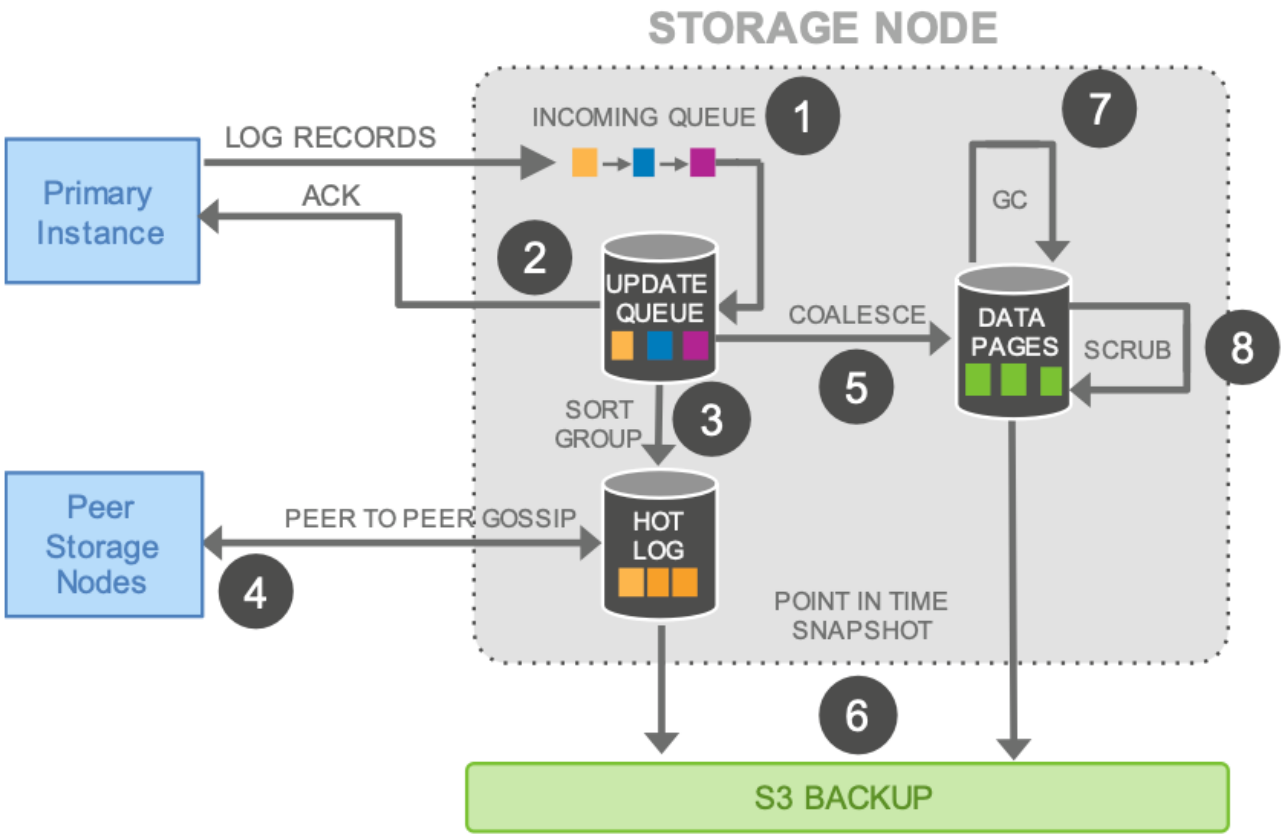

Figure 4: IO Traffic in Aurora Storage Nodes

让我们更详细地研究一下存储节点上的各种活动。如图4所示，它包括以下步骤：

（1）接收日志记录并添加到内存队列，

（2）在磁盘上保存记录并确认，

（3）组织记录并识别日志中的差异，因为某些批可能丢失，

（4）与同等角色的对比，以填补差异，

（5）将日志记录合并到新的数据页，

（6）定期将日志和新页转移到S3，

（7）定期垃圾收集旧版本，最后

（8）定期验证页面上的CRC码。

请注意，上述步骤不仅是异步的，而且前台路径中只有步骤（1）和（2）可能会影响延迟。

四、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.

在本节中，我们将描述如何从数据库引擎生成日志，以便持久状态、运行时状态和副本状态始终一致。特别是，我们将描述如何在没有昂贵的2PC协议的情况下有效地实现一致性。首先，我们将展示如何避免在崩溃恢复时进行昂贵的重做处理。接下来，我们将解释正常操作以及如何维护运行时和副本状态。最后，我们提供了恢复过程的详细信息。

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.

由于我们将数据库建模为重做日志流（如第3节所述），因此我们可以利用日志作为有序的更改序列前进的事实。实际上，每个日志记录都有一个相关联的日志序列号（LSN），它是数据库生成的单调递增值。

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.

这使我们能够通过以异步方式处理问题而不是使用像2PC这样的协议来简化维护状态的共识协议，因为2PC是健谈且不能容忍故障的。在高级别上，我们保持一致性和持久性，并在收到未完成存储请求的确认时不断提高这些点。由于任何单个存储节点都可能丢失了一条或多条日志记录，因此它们会与PG的其其它成员对齐，寻找漏洞并填补漏洞。数据库维护的运行时状态允许我们使用单段读取而不是仲裁读取，除非在恢复时状态丢失并且必须重建。

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.

数据库可能有多个未完成的独立事务，这些事务可以按与启动不同的顺序完成（达到完成和持久状态）。假设数据库崩溃或重新启动，对于每个单独的事务，是否回滚的确定是独立的。跟踪部分完成的事务并撤消它们的逻辑保存在数据库引擎中，就像它写入简单的磁盘一样。但是，在重新启动时，在允许数据库访问存储卷之前，存储服务会自行进行恢复，恢复的重点不是用户级事务，而是确保数据库能够看到统一的存储视图，尽管它是分布式的。

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.

存储服务确定其可以保证所有先前日志记录可用性的最高LSN（称为VCL或卷完整LSN）。在存储恢复期间，必须截断LSN大于VCL的每个日志记录。然而，数据库可以通过标记日志记录并将其标识为CPL或一致性点LSN来进一步约束允许截断的点子集。因此，我们将VDL或卷持久LSN定义为小于或等于VCL的最高CPL，并截断LSN大于VDL的所有日志记录。例如，即使我们拥有LSN 1007之前的完整数据，数据库也可能声明只有900、1000和1100是CPL，在这种情况下，我们必须将其截断为1000。我们完成了到1007的数据，但只能持久到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.

因此，完整性和持久性是不同的，可以将CPL视为描绘了必须按顺序接受的某种有限形式的存储系统事务。如果客户端不需要这种区分，它可以简单地将每个日志记录标记为CPL。在实践中，数据库和存储交互如下：

1.每个数据库级事务被分解为多个小型事务（MTR），这些事务是有序的，必须以原子方式执行。

2.每个小事务由多个连续日志记录组成（根据需要）。

3.小型事务中的最终日志记录是CPL。

恢复时，数据库与存储服务对话以建立每个PG的持久点，并使用该持久点建立VDL，然后发出命令以截断VDL以上的日志记录。

4.2 Normal Operation

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

我们现在描述数据库引擎的“正常操作”，并依次关注写、读、提交和副本。

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.

在Aurora中，数据库持续与存储服务交互，并维护状态以建立仲裁、提高卷的持久性，并将事务注册为已提交。例如，在正常/正向路径中，当数据库接收到确认以建立每批日志记录的写入仲裁时，它会推进当前VDL。在任何给定时刻，数据库中都可能有大量活动的并发事务，每个事务都会生成自己的重

做日志记录。数据库为每个日志记录分配一个唯一的有序LSN，该LSN受到一个约束，即没有为任何LSN分配一个大于当前VDL和一个称为LSN分配限制（LAL）的常数之和的值（当前设置为1000万）。此限制可确保数据库不会遥遥领先于存储系统，并在存储或网络无法跟上的情况下引入反向压力，从而限制传入写入。

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.

请注意，每个PG的每个段只能看到卷中影响驻留在该段上的页面的日志记录子集。每个日志记录包含一个反向链接，用于标识该PG的前一个日志记录。这些反向链接可用于跟踪已到达每个段的日志记录的完整点，以建立一个段完整LSN（SCL），该LSN标识接收到PG所有日志记录的最大LSN。存储节点在彼此闲聊时使用SCL，以查找和交换丢失的日志记录。

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.

在Aurora中，事务提交是异步完成的。当客户机提交事务时，处理提交请求的线程通过将其“提交LSN”记录为等待提交的单独事务列表的一部分，将事务放在一边，然后继续执行其他工作。与WAL协议等价的是，当且仅当最新VDL大于或等于事务的提交LSN时，基于完成提交。随着VDL的推进，数据库识别等待提交的符合条件的事务，并使用专用线程向等待提交的客户端发送提交确认。工作线程不会暂停提交，它们只是拉取其他挂起的请求并继续处理。

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.

在Aurora中，与大多数数据库一样，页面是从缓冲区缓存提供的，只有当所讨论的页面不在缓存中时，才会产生存储IO请求。

如果缓冲区缓存已满，则系统会找到要从缓存中退出的受害页面。在传统系统中，如果受害者是“脏页”，则在替换之前将其刷新到磁盘。这是为了确保页面的后续提取总是产生最新数据。虽然Aurora数据库不会在逐出时（或其他任何地方）写出页面，但它强制执行了类似的保证：缓冲区缓存中的页面必须始终是最新版本。只有当页面的“页面LSN”（标识与页面最新更改相关联的日志记录）大于或等于VDL时，才能通过从缓存中逐出页面来实现该保证。该协议确保：（a）页面中的所有更改都已在日志中硬化，（b）在缓存未命中时，请求当前VDL的页面版本以获取其最新持久版本就足够了。

在正常情况下，数据库不需要使用读取仲裁来建立共识。从磁盘读取页面时，数据库会建立一个读取点，表示发出请求时的VDL。然后，数据库可以选择一个相对于读取点完整的存储节点，知道它将因此接收最新版本。存储节点返回的页面必须与数据库中小型事务（MTR）的预期语义一致。由于数据库直接管理向存储节点馈送日志记录和跟踪进度（即每个段的SCL），因此它通常知道哪个段能够满足读取要求（SCL大于读取点的段），因此可以直接向具有足够数据的段发出读取请求。

假设数据库知道所有未完成的读取，它可以在任何时候基于每PG计算最小读取点LSN。如果存在读取副本，则写入者与它们闲聊，以在所有节点上建立每PG最小读取点LSN。该值称为保护组最小读取点LSN（PGMRPL），表示“低水位线”，低于该水位线，PG的所有日志记录都是不必要的。换句话说，存储节点段保证不会有读点低于PGMRPL的读页请求。每个存储节点都知道数据库中的PGMRPL，因此可以通过合并较旧的日志记录，然后安全地对其进行垃圾收集，来推进磁盘上的物化页面。

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.

实际的并发控制协议是在数据库引擎中执行的，就好像数据库页面和撤消段是在本地存储中组织的，就像传统的MySQL一样。

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).

在Aurora中，单个写入程序和最多15个读取副本都可以装载单个共享存储卷。因此，读取副本不会增加消耗的存储或磁盘写入操作的额外成本。为了最小化延迟，写入程序生成并发送到存储节点的日志流也会发送到所有读取副本。在阅读器中，数据库通过依次考虑每个日志记录来使用该日志流。如果日志记录引用读取器缓冲区缓存中的页面，它将使用日志应用程序将指定的重做操作应用于缓存中的该页面。否则，它只会丢弃日志记录。请注意，从编写器的角度来看，复制副本异步使用日志记录，这承认用户提交与复制副本无关。复制副本在应用日志记录时遵守以下两条重要规则：（a）将应用的唯一日志记录是LSN小于或等于VDL的日志记录，以及（b）作为单个迷你事务一部分的日志记录在复制副本的缓存中原子应用，以确保复制副本看到所有数据库对象的一致视图。在实践中，每个副本通常落后于写入程序一个短间隔（20ms或更短）。

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.

大多数传统数据库使用ARIES[7]等恢复协议，该协议取决于是否存在可表示所有提交事务的精确内容的预写日志（WAL）。这些系统还定期检查数据库，通过将脏页刷新到磁盘并将检查点记录写入日志，以粗粒度方式建立持久性点。在重新启动时，任何给定页面都可能丢失一些已提交的数据或包含未提交的数据。因此，在崩溃恢复时，系统使用日志应用程序将每个日志记录应用到相关数据库页面，从而处理自上次检查点以来的重做日志记录。此过程使数据库页面在故障点处于一致状态，之后，可以通过执行相关的撤消日志记录来回滚崩溃期间的运行中事务。故障恢复可能是一项昂贵的操作。减少检查点间隔有帮助，但要以干扰前台事务为代价。Aurora不需要这样的权衡。

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.

传统数据库的一个重要简化原则是，在前向处理路径和恢复路径中使用相同的重做日志应用程序，在数据库脱机时，该应用程序在前台同步运行。我们在Aurora中也依赖同样的原理，除了重做日志应用程序与数据库解耦，并在存储节点上并行操作，并且始终在后台操作。一旦数据库启动，它将与存储服务协作执行卷恢复，因此，即使在每秒处理100000多条写语句时崩溃，Aurora数据库也可以非常快速地恢复（通常不到10秒）

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.

数据库确实需要在崩溃后重建其运行时状态。在这种情况下，它会为每个PG联系段的读仲裁，这足以保证发现可能已达到写仲裁的任何数据。一旦数据库为每个PG建立了读仲裁，它就可以通过生成截断范围来重新计算VDL，在该截断范围之上，数据被截断，该截断范围在新的VDL之后将每个日志记录作废，直到并包括一个结束LSN，数据库可以证明该结束LSN至少与可能看到的最高未完成日志记录一样高。数据库推断这个上限是因为它分配LSN，并限制了VDL（前面描述的1000万限制）以上的分配距离。截断范围使用历元编号进行版本控制，并持久地写入存储服务，以便在恢复中断和重新启动的情况下，不会混淆截断的持久性。

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.

数据库仍然需要执行撤消恢复，以在崩溃时解除正在进行的事务的操作。但是，当系统从撤消段构建这些正在进行的事务的列表后，数据库联机时，可以进行撤消恢复。

五、 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.

在本节中，我们描述了Aurora的构建块，如图5中的鸟瞰图所示。

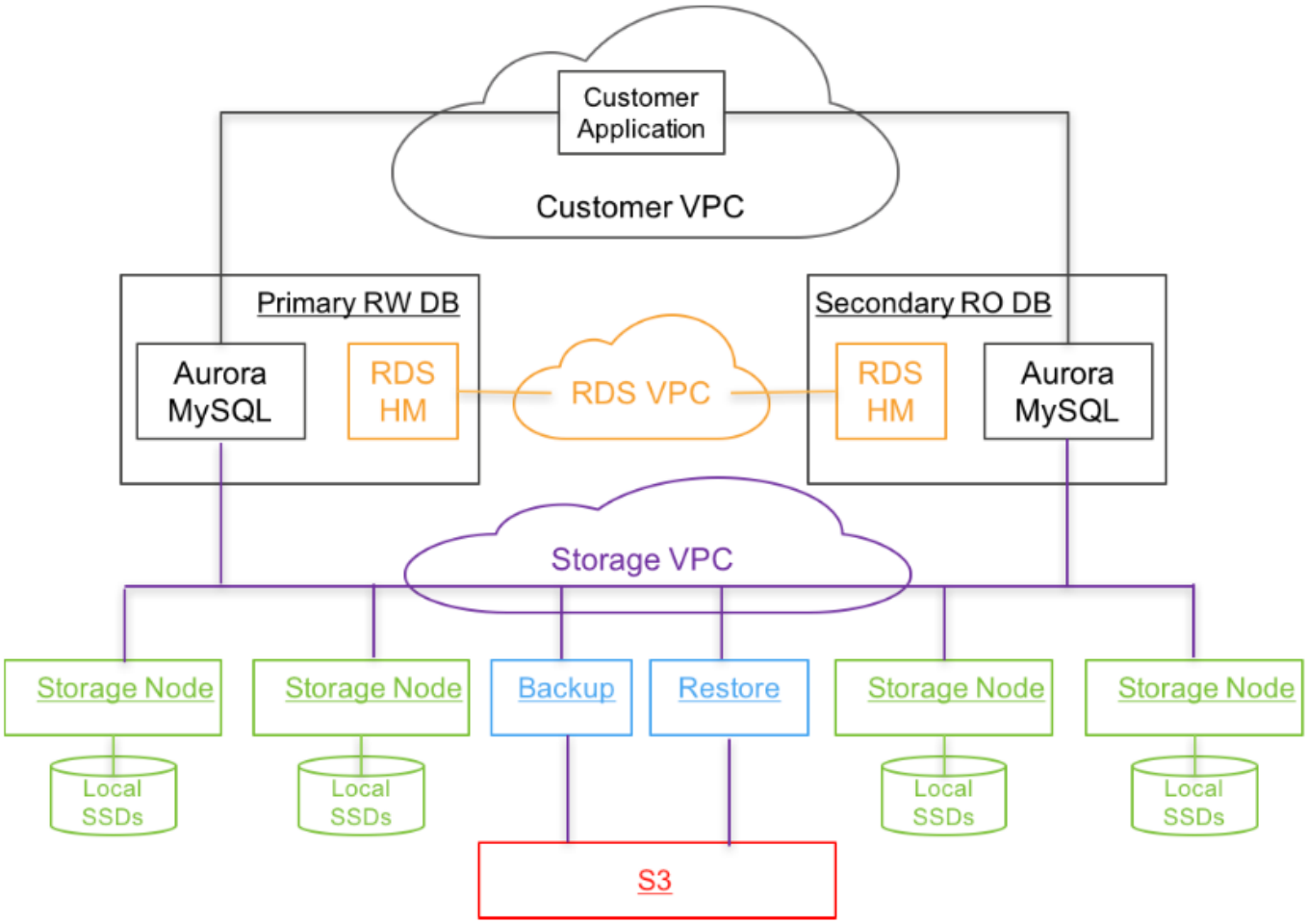

Figure 5: Aurora Architecture: A Bird's Eye View

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).

数据库引擎是“社区”MySQL/InnoDB的分支，主要在InnoDB如何将数据读写到磁盘上。在community InnoDB中，写入操作会导致缓冲区页面中的数据被修改，相关的重做日志记录以LSN顺序写入WAL的缓冲区。在事务提交时，WAL协议仅要求将事务的重做日志记录持久地写入磁盘。实际修改的缓冲区页面也最终通过双写技术写入磁盘，以避免部分页面写入。这些页面写入发生在后台，或从缓存中逐出时，或在执行检查点时。除了IO子系统之外，InnoDB还包括事务子系统、锁管理器、

B+树实现和相关的“迷你事务”（MTR）概念。MTR是一种仅在InnoDB内部使用的构造，它对必须以原子方式执行的操作组进行建模（例如，拆分/合并B+树页面）。

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.

在Aurora InnoDB变体中，表示必须在每个MTR中原子执行的更改的重做日志记录被组织成批，这些批由每个日志记录所属的PGs分片，这些批被写入存储服务。每个MTR的最终日志记录被标记为一致性点。Aurora支持与社区MySQL在编写器中完全相同的隔离级别（标准ANSI级别和快照隔离或一致读取）。Aurora read副本在编写器中获取有关事务开始和提交的连续信息，并使用这些信息来支持本地事务的快照隔离，这些事务当然是只读的。请注意，并发控制完全在数据库引擎中实现，不会影响存储服务。存储服务提供了底层数据的统一视图，在逻辑上与将数据写入community InnoDB中的本地存储所获得的相同。

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.

Aurora利用Amazon关系数据库服务（RDS）作为其控制平面。RDS在数据库实例上包括一个名为主机管理器（HM）的代理，它监视集群的运行状况，并确定是否需要故障转移，或者是否需要替换实例。每个数据库实例都是由单个写入器和零个或多个读取副本组成的集群的一部分。集群的实例位于单个地理区域（例如，美国东部-1、美国西部-1等），通常放置在不同的AZ中，并连接到同一区域的存储车队。为了安全起见，我们隔离了数据库、应用程序和存储之间的通信。实际上，每个数据库实例可以在三个亚马逊虚拟专用云（VPC）网络上通信：客户应用程序与引擎交互的客户专有网络、数据库引擎与控制平面交互的RDS专有网络，以及数据库与存储服务交互的存储专有网络。

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.

存储服务部署在EC2虚拟机集群上，这些虚拟机跨每个区域中的至少3个AZ进行调配，并共同负责调配多个客户存储卷，向这些卷读取和写入数据，以及从这些卷备份和恢复数据。存储节点操作本地SSD，并与数据库引擎实例、其他对等存储节点和备份/恢复服务交互，这些服务将更改的数据连续备份到S3，并根据需要从S3恢复数据。存储控制平面使用Amazon DynamoDB数据库服务来持久存储群集和存储卷配置、卷元数据以及备份到S3的数据的详细描述。为了协调长时间运行的操作，例如数据库卷恢复操作或存储节点故障后的修复（重新复制）操作，存储控制平面使用Amazon简单工作流服务。保持高级别的可用性需要在最终用户受到影响之前，主动、自动和早期地检测实际和潜在问题。使用指标收集服务不断监控存储操作的所有关键方面，如果关键性能或可用性指标表明存在问题，则会发出警报。

六、 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.

在本节中，我们将分享我们将Aurora作为生产服务运行的经验，该服务于2015年7月“普遍可用”。我们首先总结运行行业标准基准的结果，然后介绍客户的一些性能结果。

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.

在这里，我们展示了不同实验的结果，这些实验使用行业标准基准（如SysBench和TPC-C变体）比较了Aurora和MySQL的性能。我们在连接到EBS卷的实例上运行MySQL，该卷的IOPS为30K。除非另有说明，这些都是r3.8X大型EC2实例，具有32个VCPU和244GB内存，并采用英特尔至强E5-2670 v2（常春藤桥）处理器。r3.8XL上的缓冲区缓存设置为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.

在本实验中，我们报告了Aurora中的吞吐量可以随实例大小线性扩展，最高实例大小可以是MySQL 5.6和MySQL 5.7的5倍。请注意，Aurora目前基于MySQL 5.6代码库。我们在r3系列的5个EC2实例（大型、xlarge、2xlarge、4xlarge、8xlarge）上为1GB数据集（250个表）运行了SysBench只读和只写基准测试。每个实例大小正好是紧接着更大实例的VCPU和内存的一半。

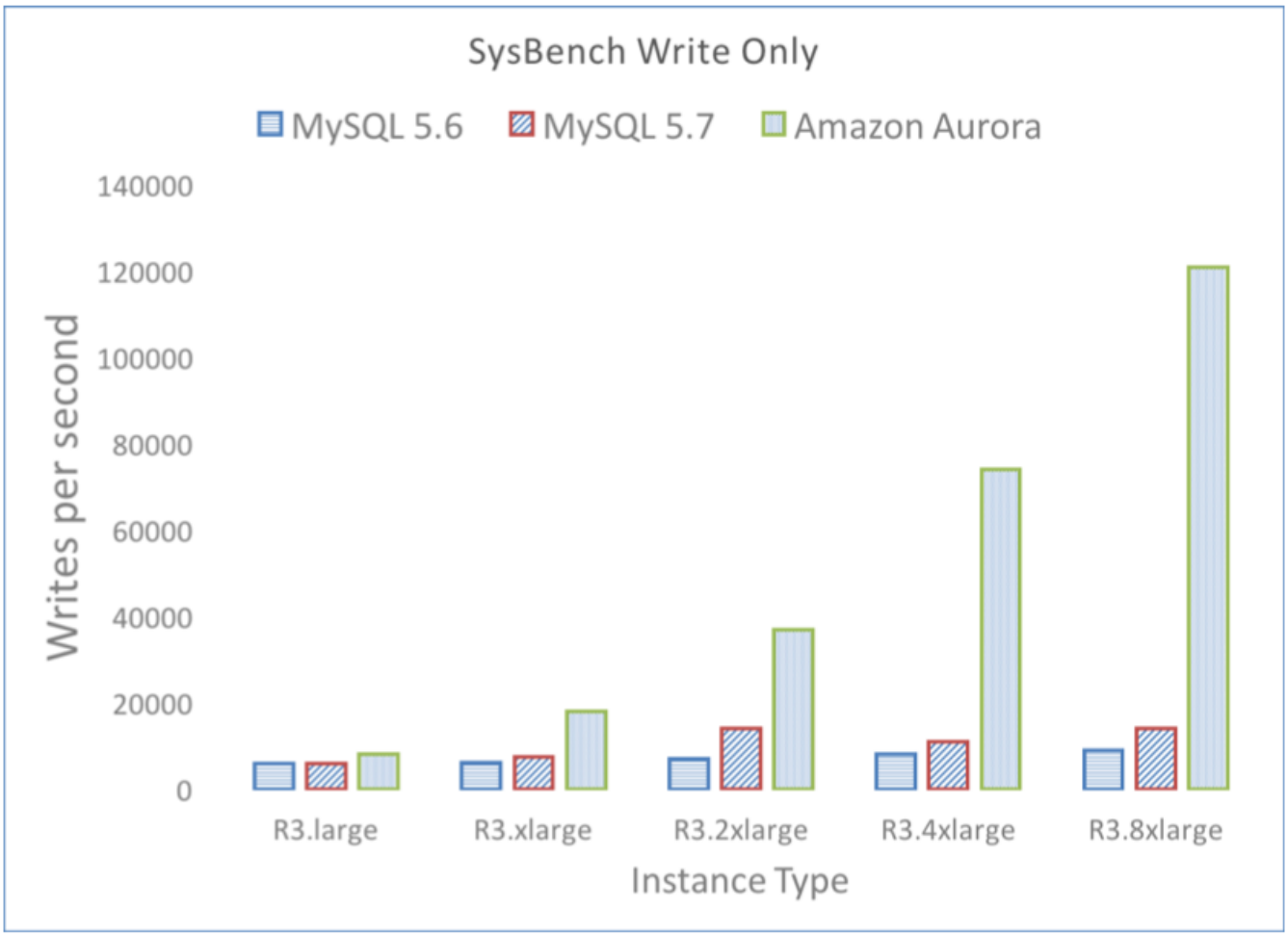

Figure 7: Aurora scales linearly for write-only workload

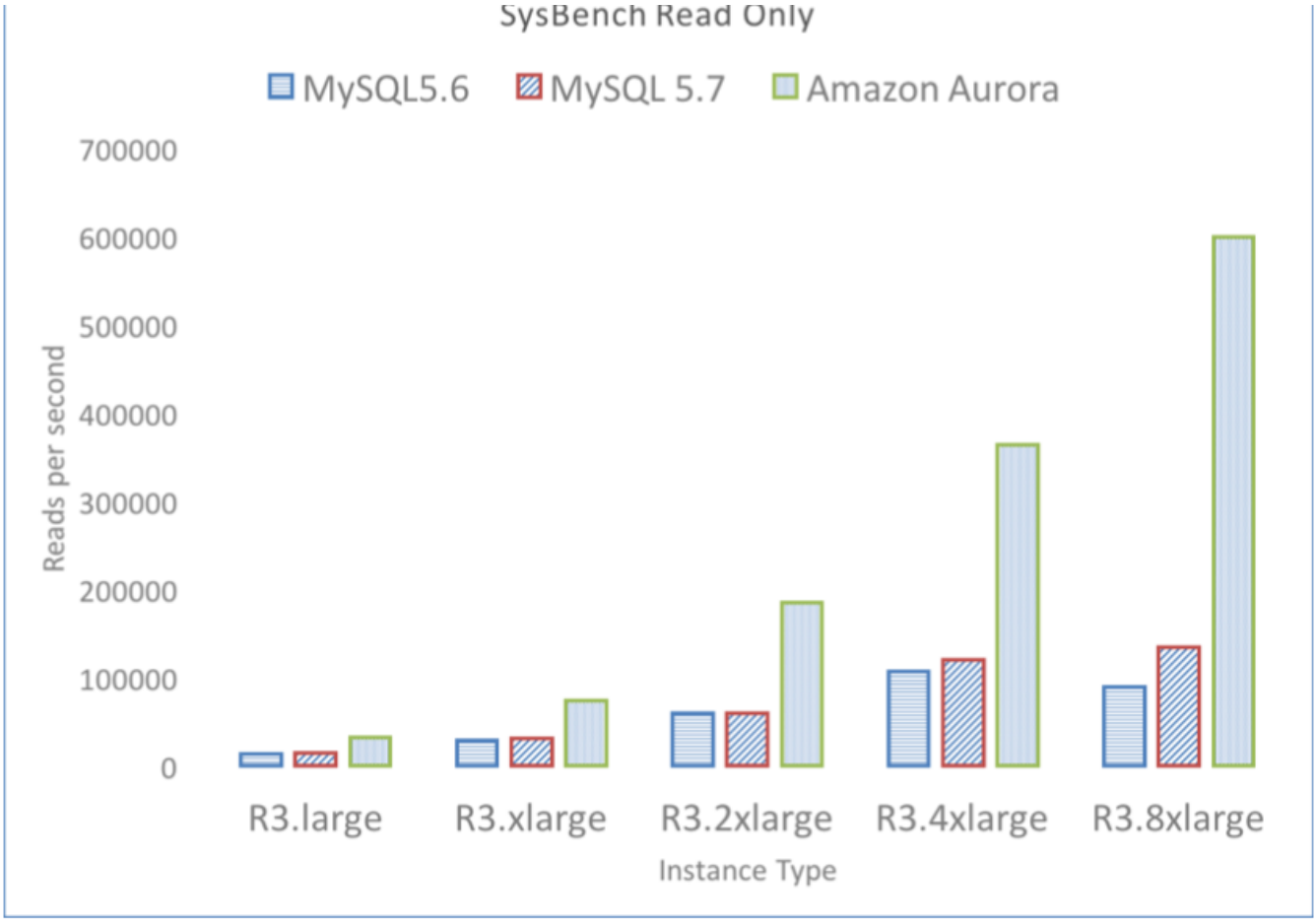

Figure 6: Aurora scales linearly for read-only workload

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.

结果如图7和图6所示，分别以每秒写语句和读语句来衡量性能。对于每个更高的实例大小，Aurora的性能都会翻一番，对于r3.8xlarge，它实现了121000次写入/秒和600000次读取/秒，是MySQL 5.7的5倍，其最高取速度为20000次/秒读取和125000次/秒写入。

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.

在这个实验中，我们报告说，即使在数据量较大的情况下，包括缓存外工作集的工作负载，Aurora的吞吐量也大大超过了MySQL。表2显示，对于SysBench只写工作负载，Aurora比数据库大小为100GB的MySQL快67倍。即使对于1TB的数据库大小和缓存外的工作负载，Aurora仍然比MySQL快34倍

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.

在本实验中，我们报告了Aurora中的吞吐量可以随着客户端连接的数量而增加。表3显示了在连接数从50到500到5000的过程中，以写/秒为单位运行SysBench OLTP基准测试的结果。虽然Aurora从40000写/秒扩展到110000写/秒，但MySQL的吞吐量在大约500个连接处达到峰值，然后随着连接数增加到5000，吞吐量急剧下降。

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.

在本实验中，我们报告说，即使在更高的工作负载下，Aurora读取副本的延迟也明显低于MySQL副本。表4显示，当工作负载从1000到10000次写入/秒变化时，Aurora中的副本延迟从2.62毫秒增长到5.38毫秒。相反，MySQL中的副本延迟从不到一秒增长到300秒。在每秒10000次写入时，Aurora的副本延迟比MySQL的要小几个数量级。副本延迟是根据提交的事务在副本中可见所需的时间来衡量的。

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.

在本实验中，我们报告说，相对于MySQL，Aurora在具有热行争用的工作负载（例如基于TPC-C基准的工作负载）上表现得非常好。我们在r3.8xlarge上运行了针对Amazon Aurora和MySQL 5.6和5.7的Percona TPC-C变体[37]，其中MySQL使用具有30K配置IOPS的EBS卷。表5显示，当工作负载从500个连接和10GB数据大小变化到5000个连接和100GB数据大小时，Aurora可以维持MySQL 5.7的2.3倍到16.3倍的吞吐量。

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.

在本节中，我们将分享一些将生产工作负载从MySQL迁移到Aurora的客户报告的结果。

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.

一家互联网游戏公司在r3.4xlarge实例上将其生产服务从MySQL迁移到Aurora。在迁移之前，他们的web事务所经历的平均响应时间是15毫秒。相反，在迁移之后，平均响应时间为5.5毫秒，改进了3倍，如图8所示。

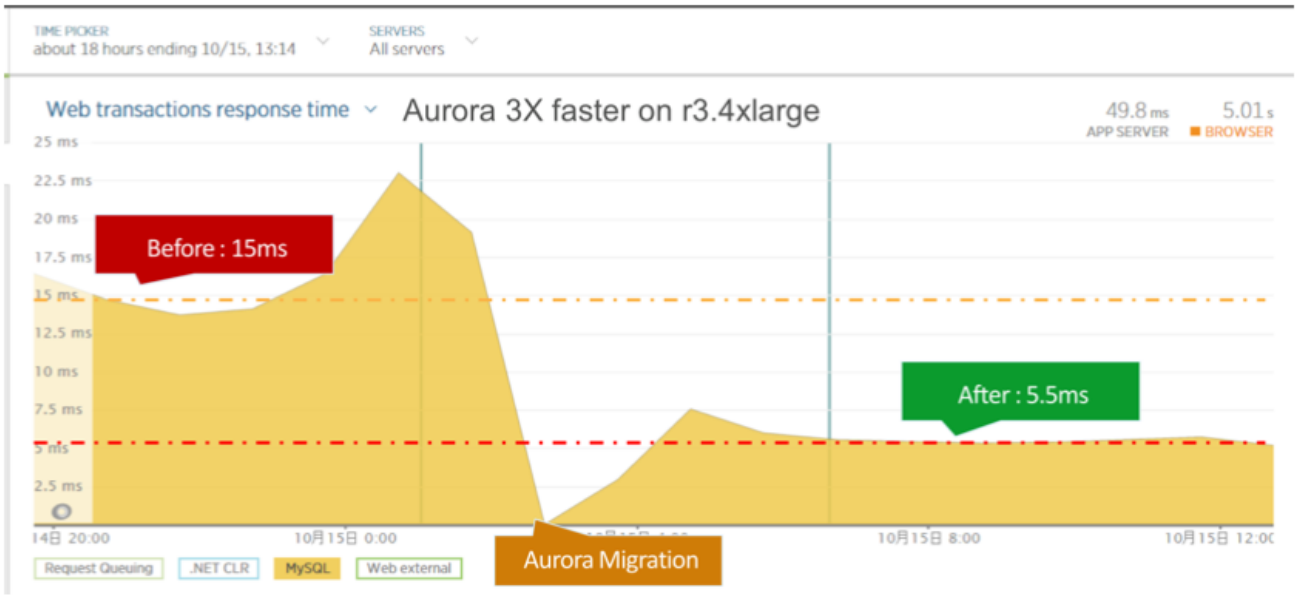

Figure 8: Web application response time

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.

一家教育技术公司，其服务帮助学校管理学生笔记本电脑，将其生产工作量从MySQL迁移到了Aurora。图9和图10显示了迁移前后（14:00时）选择和每记录插入操作的中值（P50）和第95百分位（P99）延迟。

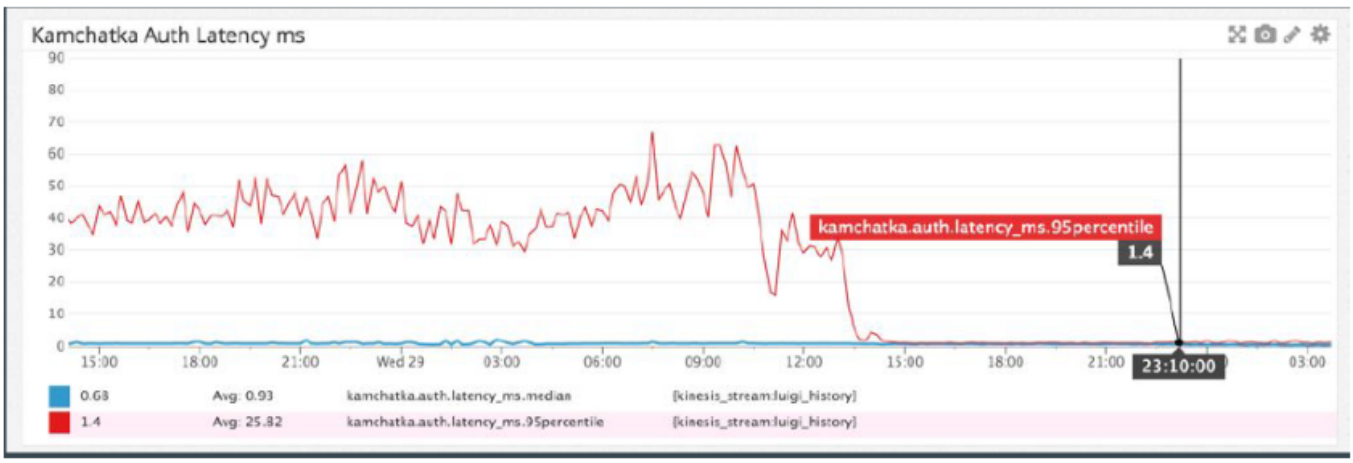

Figure 9: SELECT latency (P50 vs P95)

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.

迁移前，P95潜伏期介于40ms至80ms之间，远低于P50潜伏期约1ms。应用程序正经历着我们在本文前面描述的那种糟糕的异常值性能。然而，在迁移之后，这两种操作的P95延迟显著改善，并接近P50延迟。

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.

正如Weiner在Pinterest[40]上所报告的那样，MySQL副本常常远远落后于其作者，并且可能“导致奇怪的错误”。对于前面描述的教育技术公司，副本延迟通常会达到12分钟，并影响应用程序的正确性，因此副本只能作为备用。相反，在迁移到Aurora之后，4个副本之间的最大副本延迟从未超过20毫秒，如图11所示。Aurora提供的改进的副本延迟使公司能够将其应用程序负载的很大一部分转移到副本上，从而节省成本并提高可用性。

Figure 11: Maximum Replica Lag (averaged hourly)

七、 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.

现在，我们已经看到客户运行的各种应用程序，从小型互联网公司一直到运营大量Aurora集群的高度复杂的组织。虽然它们的许多用例都是标准的，但我们关注的是云中常见的场景和期望，并将我们引向新的方向

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.

我们的许多客户都经营软件即服务（SaaS）业务，要么是独家经营，要么是与一些剩余的本地客户合作，他们正试图转向他们的SaaS模式。我们发现，这些客户通常依赖于他们无法轻松更改的应用程序。因此，他们通常使用模式/数据库作为租用单元，将不同的客户合并到单个实例上。这种习惯用法降低了成本：当所有客户不可能同时处于活动状态时，他们避免为每个客户支付专用实例的费用。例如，我们的一些SaaS客户报告自己拥有超过50000名客户。

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.

该模型与Salesforce等著名的多租户应用程序有明显不同。com[14]，其使用多租户数据模型，并将多个客户的数据打包到单个模式的统一表中，以每行为基础识别租户。因此，我们看到许多客户使用包含大量表的合并数据库。小型数据库的150000多个表的生产实例非常常见。这给管理元数据（如字典缓存）的组件带来了压力。更重要的是，这些客户需要（a）维持高水平的吞吐量和许多并发用户连接，（b）一种仅在使用数据时提供数据并支付数据的模型，因为很难提前预测需要多少存储空间，以及（c）减少抖动，以使单个租户的峰值对其他租户的影响最小。Aurora支持这些属性，非常适合此类SaaS应用程序。

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.

Internet工作负载通常需要处理基于突发事件的流量峰值。我们的一个主要客户在一个非常受欢迎的全国电视节目中特别露面，并经历了一次这样的峰值，大大超过了正常的峰值吞吐量，而不会对数据库造成压力。为了支持这种峰值，数据库处理多个并发连接非常重要。这种方法在Aurora中是可行的，因为底层存储系统扩展得非常好。我们有几个客户每秒运行8000多个连接。

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.

Ruby on Rails等现代web应用程序框架深入集成了对象-关系映射工具。因此，应用程序开发人员很容易对其数据库进行许多模式更改，这使得DBA难以管理模式的演变。在Rails应用程序中，这些被称为“DB迁移”，我们听到了DBA的第一手资料，他们要么“每周处理几十次迁移”，要么制定对冲策略，以确保未来的迁移顺利进行。MySQL提供了自由的模式演化语义，并使用完整的表副本实现了大多数更改，这使得情况更加恶化。由于频繁的DDL是一种实用的现实，我们已经实现了一种高效的在线DDL实现，它（a）基于每页对模式进行版本，并使用其模式历史按需解码各个页面，以及（b）使用写时修改原语将各个页面延迟升级到最新模式。

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.

我们的客户对云原生数据库有着苛刻的期望，这可能与我们如何运营车队以及我们修补服务器的频率相冲突。由于我们的客户主要将Aurora用作支持生产应用程序的OLTP服务，因此任何中断都可能造成创伤。因此，我们的许多客户对我们的数据库软件更新的容忍度非常低，即使这相当于每6周左右30秒的计划停机时间。因此，我们最近发布了一个新的零停机补丁（ZDP）功能，允许我们在不影响飞行中数据库连接的情况下对客户进行补丁。

如图12所示，ZDP通过查找没有活动事务的瞬间来工作，并在该瞬间将应用程序状态假脱机到本地临时存储，修补引擎，然后重新加载应用程序状态。在这个过程中，用户会话保持活动状态，并且不知

道引擎在幕后发生了变化。

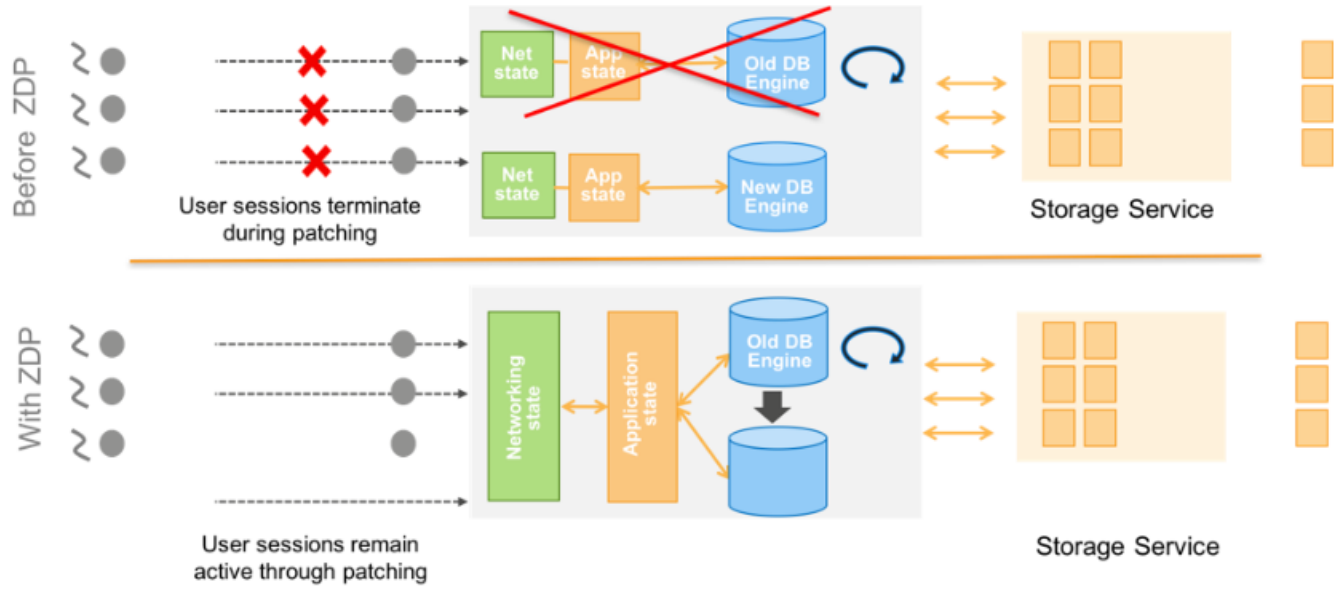

Figure 12: Zero-Downtime Patching

八、RELATED WORK

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

在本节中，我们将讨论其他贡献，以及它们与Aurora中采取的方法的关系。

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.

将存储与计算解耦。虽然传统的系统通常被构建为单片守护进程[27]，但最近有一些数据库将内核分解为不同的组件。例如，Deuteronomy[10]就是这样一个系统，它将提供并发控制和恢复的事务组件（TC）与在LLAMA[34]之上提供访问方法的数据组件（DC）分离开来，LLAMA是一个无锁日志结构的缓存和存储管理器。Sinfonia[39]和Hyder[38]是通过扩展服务抽象事务访问方法的系统，数据库系统可以使用这些抽象来实现。Yesquel[36]系统实现了多版本分布式平衡树，并将并发控制与查询处理器分离。Aurora以低于《申命记》、《海德》、《辛福尼亚》和《耶斯克尔》的水平将存储解耦。在

Aurora中，查询处理、事务、并发、缓冲区缓存和访问方法与日志记录、存储和恢复分离，并作为扩展服务实现。

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.

分布式系统。分区的正确性和可用性之间的权衡早已为人所知，其主要结果是，在网络分区的情况下，单拷贝串行化是不可能的[15]。最近在[16]中证明的布鲁尔CAP定理指出，在存在网络分区的情况下，高可用性系统无法提供“强”一致性保证。这些结果以及我们在云级复杂和相关故障方面的经验推动了我们的一致性目标，即使存在由AZ故障导致的分区。

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.

Bailis等人[12]研究了提供高可用事务（HATs）的问题，这些事务既不会在分区期间不可用，也不会导致高网络延迟。它们表明，串行化、快照隔离和可重复读取隔离不符合HAT，而大多数其他隔离级别都可以通过高可用性实现。Aurora通过简化假设提供了所有这些隔离级别，即在任何时候都只有一个编写器使用从单个有序域分配的LSN生成日志更新。

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.

谷歌的Spanner[24]提供了外部一致的[25]读写，以及在一个时间戳上跨数据库的全局一致读写。这些特性使SPanner能够支持一致的备份、一致的分布式查询处理[26]和原子模式更新，所有这些都是全局范围的，甚至在存在正在进行的事务的情况下。正如Bailis[12]所解释的，SPanner非常专门用于Google的读重工作负载，并依赖于读/写事务的两阶段提交和两阶段锁定。

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.

并发控制。较弱的一致性（PACELC[17]）和隔离模型[18][20]在分布式数据库中是众所周知的，并导致了乐观复制技术[19]以及最终的一致性系统[21][22][23]。集中式系统中的其他方法包括基于锁定的经典悲观方案[28]、Hekaton[29]中的多版本并发控制等乐观方案、VoltDB[30]和HyPer[31][32]和Deuteronomy中的时间戳排序等分片方法。Aurora的存储服务为数据库引擎提供了持久持久化的本地磁盘的抽象，并允许引擎确定隔离和并发控制。

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.

日志是结构化存储。日志结构存储系统是由LFS[33]于1992年引入的。最近的《申命记》以及LLAMA[34]和Bw Tree[35]中的相关工作在存储引擎堆栈中以多种方式使用日志结构技术，并像Aurora一样，通过写增量而不是整页来减少写放大。Deuteronomy和Aurora都实现了纯重做日志记录，并跟踪用于确认提交的最高稳定LSN。

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.

恢复虽然传统数据库依赖基于ARIES的恢复协议[5]，但最近的一些系统选择了其他性能路径。例如，Hekaton和VoltDB在崩溃后使用某种形式的更新日志重建其内存状态。Sinfonia[39]等系统通过使用进程对和状态机复制等技术避免恢复。Graefe[41]描述了一种具有每页日志记录链的系统，该系统支持按需逐页重做，可以快速恢复。像极光一样，申命记不需要重做或恢复。这是因为《申命记》延迟了事务，因此只有提交的更新才会发布到持久存储中。因此，与极光不同，在《申命记》中，交易的规模可以受到限制。

九、 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.

我们将Aurora设计为一个高吞吐量OLTP数据库，在云规模环境中既不损害可用性也不损害耐用性。大的想法是摆脱传统数据库的整体架构，将存储与计算分离。特别是，我们将数据库内核的下四分之一移动到一个独立的可扩展分布式服务，该服务管理日志记录和存储。由于所有I/O都是通过网络编写的，我们的基本约束现在是网络。因此，我们需要关注减轻网络负担和提高吞吐量的技术。我们依赖仲裁模型来处理大规模云环境中发生的复杂和相关故障，并避免离群性能损失、日志处理以减少总I/O负担、异步共识以消除多阶段同步协议、离线崩溃恢复和分布式存储中的检查点。我们的方法简化了架构，降低了复杂性，易于扩展，并为未来的发展奠定了基础。

十、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.

我们感谢整个极光开发团队在该项目上的努力，包括我们的现任成员以及我们杰出的校友（詹姆斯·科里、萨姆·麦凯维、扬·莱欣斯基、隆·隆德格伦、普拉迪普·马哈瓦拉普和斯特凡诺·斯特凡尼）。我们特别感谢使用我们的服务操作生产工作负载的客户，并慷慨地与我们分享他们的经验和期望。我们还感谢牧羊人在形成本文时提出的宝贵意见。

十一、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.

.....

[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.