Dynamo:亚马逊高度可用的键值对存储

Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman、Alex Pilchin、Swaminathan Sivasubramanian、Peter Vosshall 和 Werner Vogels

亚马逊网站

摘要大规模的可靠

性是我们在亚马逊面临的最大挑战之一,亚马逊是世界上最大的电子商务运营商之一。即使是最轻微的中断也会产生重大的财务后果并影响客户信任。为全球许多网站提供服务的 Amazon.com 平台是在位于全球许多数据中心的数万台服务器和网络组件的基础设施之上实施的。在这种规模下,小型和大型组件会不断发生故障,并且面对这些故障时管理持久状态的方式推动了软件系统的可靠性和可扩展性。

本文介绍了 Dynamo 的设计和实现,这是一个高度可用的键值存储系统,Amazon 的一些核心服务使用它来提供“永远在线”的体验。为了实现这一级别的可用性,Dynamo 在某些故障场景下牺牲了一致性。它广泛使用对象版本控制和应用程序辅助的冲突解决,为开发人员提供了一个新颖的界面供使用。

类别和主题描述符

D.4.2 [操作系统]:存储管理; D.4.5 [操作系统]:可靠性; D.4.2 [操作系统]:性能;

一般条款

算法、管理、测量、性能、设计、可靠性。

1. 引言亚马逊运营着一个全球电

子商务平台,在高峰时间使用位于世界各地许多数据中心的数万台服务器为数千万客户提供服务。亚马逊平台在性能、可靠性和效率方面有严格的运营要求,为了支持持续增长,平台需要具有高度可扩展性。可靠性是最重要的要求之一,因为即使是最轻微的中断也会产生重大的财务后果并影响客户信任。此外,为了支持持续增长,该平台需要具有高度可扩展性。

SOSP 07,2007 年10 月 14-17 日,美国华盛顿史蒂文森。

我们的组织从运营 Amazon 平台中学到的教训之一是,系统的可靠性和可扩展性取决于其应用程序状态的管理方式。

Amazon 使用由数百个服务组成的高度分散、松散耦合、面向服务的架构。在这种环境中,特别需要始终可用的存储技术。例如,即使磁盘出现故障、网络路由抖动或数据中心被龙卷风摧毁,客户也应该能够查看商品并将其添加到他们的购物车中。因此,负责管理购物车的服务要求它始终可以写入和读取其数据存储,并且它的数据需要跨多个数据中心可用。

处理由数百万个组件组成的基础设施中的故障是我们的标准操作模式;在任何给定时间,总是有少量但大量的服务器和网络组件发生故障。因此,亚马逊的软件系统需要以将故障处理视为正常情况而不影响可用性或性能的方式构建。

为了满足可靠性和扩展需求,Amazon 开发了多种存储技术,其中最著名的可能是 Amazon Simple Storage Service(也可在 Amazon 之外使用,称为 Amazon S3)。本文介绍了 Dynamo 的设计和实现,Dynamo 是为 Amazon 平台构建的另一个高度可用和可扩展的分布式数据存储。

Dynamo 用于管理具有非常高的可靠性要求并且需要严格控制可用性、一致性、成本效益和性能之间的权衡的服务状态。 Amazon 的平台有一组非常多样化的应用程序,它们具有不同的存储要求。一组选定的应用程序需要一种足够灵活的存储技术,以使应用程序设计人员能够根据这些折衷适当地配置其数据存储,从而以最具成本效益的方式实现高可用性和保证性能。

亚马逊平台上有许多服务只需要对数据存储进行主键访问。对于许多服务,例如提供畅销列表、购物车、客户偏好、会话管理、销售排名和产品目录的服务,使用关系数据库的常见模式会导致效率低下并限制规模和可用性。 Dynamo 提供了一个简单的仅主键接口来满足这些应用程序的要求。

Dynamo 综合了众所周知的技术来实现可扩展性和可用性:使用一致的散列 [10] 对数据进行分区和复制,并且通过对象版本控制 [12] 促进一致性。更新期间副本之间的一致性通过类似仲裁的技术和分散的副本同步协议来维护。迪纳摩雇用

基于 gossip 的分布式故障检测和成员协议。 Dynamo 是一个完全去中心化的系统,几乎不需要手动管理。存储节点可以在 Dynamo 中添加和删除,无需任何手动分区或重新分配。

在过去的一年里,Dynamo 一直是亚马逊电子商务平台中多项核心服务的底层存储技术。在繁忙的假日购物季节,它能够有效地扩展到极端峰值负载,而无需任何停机。例如,维护购物车的服务（Shopping Cart Service)处理了数千万个请求,导致单日超过 300 万次结账,而管理会话状态的服务处理了数十万个并发活动会话。

这项工作对研究界的主要贡献是评估如何组合不同的技术以提供单一的高可用性系统。它表明最终一致的存储系统可用于要求苛刻的应用程序的生产。它还提供了对这些技术的调整的洞察,以满足具有非常严格的性能要求的生产系统的要求。

论文结构如下。第 2 节介绍了背景,第 3 节介绍了相关工作。第 4 节介绍了系统设计,第 5 节介绍了实现。第 6 节详细介绍了在生产环境中运行 Dynamo 获得的经验和见解,第 7 节总结了本文。本文中有许多地方可能需要提供更多信息,但为了保护亚马逊的商业利益,我们需要减少一定程度的细节。出于这个原因,第 6 节中的数据中心内和数据中心间延迟、第 6.2 节中的绝对请求率以及第 6.3 节中的中断长度和工作负载是通过聚合度量而不是绝对详细信息提供的。

2. 背景亚马逊的电子商务平台

由数百种服务组成,这些服务协同工作以提供从推荐到订单履行再到欺诈检测的功能。每个服务都通过定义明确的接口公开,并可通过网络访问。这些服务托管在一个基础架构中,该基础架构由遍布全球许多数据中心的数万台服务器组成。其中一些服务是无状态的（即,聚合来自其他服务的响应的服务）,而另一些是有状态的（即,通过对存储在持久存储中的状态执行业务逻辑来生成其响应的服务）。

传统上,生产系统将其状态存储在关系数据库中。然而,对于状态持久性的许多更常见的使用模式,关系数据库是一种远非理想的解决方案。这些服务中的大多数仅通过主键存储和检索数据,不需要 RDBMS 提供的复杂查询和管理功能。这种多余的功能需要昂贵的硬件和高技能的人员来操作,这使其成为一种非常低效的解决方案。

此外,可用的复制技术是有限的,通常选择一致性而不是可用性。尽管近年来取得了许多进步,但要横向扩展数据库或使用智能分区方案进行负载平衡仍然不容易。

本文介绍了 Dynamo,这是一种高度可用的数据存储技术,可满足这些重要服务类别的需求。 Dynamo 具有简单的键/值接口,具有明确定义的一致性窗口的高可用性,资源使用效率高,并且具有简单的横向扩展方案来解决数据集大小或请求率的增长。每个使用 Dynamo 的服务都运行自己的 Dynamo 实例。

2.1 系统假设和要求此类服务的存储系统有以下要求：

查询模型:对由键唯一标识的数据项进行简单的读写操作。状态存储为由唯一键标识的二进制对象（即,blob）。没有操作跨越多个数据项,也不需要关系模式。

这个要求是基于观察到亚马逊的大部分服务可以使用这个简单的查询模型并且不需要任何关系模式。 Dynamo 针对需要存储相对较小（通常小于 1 MB)对象的应用程序。

ACID 属性： ACID （原子性、一致性、隔离性、持久性)是一组属性,可确保可靠地处理数据库事务。在数据库的上下文中,对数据的单个逻辑操作称为事务。

Amazon 的经验表明,提供 ACID 保证的数据存储往往可用性较差。这已得到业界和学术界的广泛认可[5]。

如果这会导致高可用性,那么 Dynamo 会针对以较弱的一致性（ACID 中的“C”）运行的应用程序。 Dynamo 不提供任何隔离保证并且只允许单键更新。

效率:系统需要在商品硬件基础设施上运行。在亚马逊的平台中,服务具有严格的延迟要求,通常在分布的 99.9% 处测量。鉴于状态访问在服务操作中起着至关重要的作用,存储系统必须能够满足如此严格的 SLA（见下文第 2.2 节）。服务必须能够配置 Dynamo,以便它们始终如一地满足其延迟和吞吐量要求。

权衡是在性能、成本效率、可用性和耐用性保证方面。

其他假设:Dynamo 仅供亚马逊内部服务使用。它的运行环境被假定为非敌对的,并且没有认证和授权等安全相关要求。此外,由于每个服务都使用其独特的 Dynamo 实例,因此其初始设计目标是多达数百个存储主机。我们将在后面的部分讨论 Dynamo 的可扩展性限制以及可能的可扩展性相关扩展。

2.2 服务水平协议 (SLA)

为了保证应用程序可以在有限的时间内交付其功能,平台中的每个依赖项都需要以更严格的边界交付其功能。客户和服务参与服务水平协议 (SLA),这是一份正式协商的合同,其中客户和服务就几个与系统相关的特征达成一致,其中最突出的包括客户对特定 API 的预期请求率分布和预期的服务延迟在那些条件下。一个简单 SLA 的示例是一项服务,它保证它将

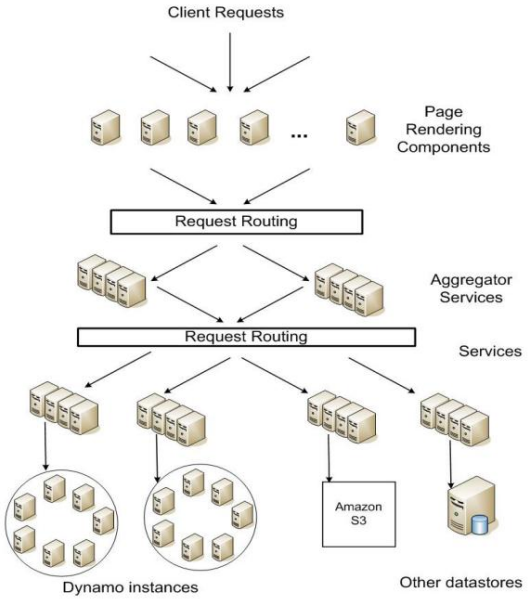

图1:亚马逊平台面向服务的架构

在每秒 500 个请求的峰值客户端负载下,为 99.9% 的请求在 300 毫秒内提供响应。

在亚马逊分散的面向服务的基础设施中,SLA 发挥着重要作用。例如,对电子商务站点之一的页面请求通常需要渲染引擎通过向 150 多个服务发送请求来构建其响应。

这些服务通常具有多个依赖关系,这些依赖关系通常是其他服务,因此应用程序的调用图具有多个级别的情况并不少见。为了确保页面渲染引擎能够在页面交付上保持明确的界限,调用链中的每个服务都必须遵守其性能契约。

图 1 显示了亚马逊平台架构的抽象视图,其中动态 Web 内容由页面渲染组件生成,这些组件依次查询许多其他服务。服务可以使用不同的数据存储来管理其状态,并且这些数据存储只能在其服务边界内访问。一些服务充当聚合器,通过使用其他几个服务来产生复合响应。通常,聚合器服务是无状态的,尽管它们使用大量缓存。

行业中形成面向性能的 SLA 的常用方法是使用平均值、中值和预期方差来描述它。在亚马逊,我们发现如果目标是构建一个所有客户(而不仅仅是大多数客户)都拥有良好体验的系统,那么这些指标还不够好。例如,如果使用广泛的个性化技术,则历史较长的客户需要更多的处理,这会影响分发高端的性能。以平均或中值响应时间表示的 SLA 将无法解决这一重要客户群的绩效。为了解决这个问题,在亚马逊,SLA 以分布的 99.9% 表示和衡量。选择 99.9% 而不是更高的百分位数是基于成本效益分析,该分析表明成本显着增加以提高性能。使用亚马逊的经验生产系统表明,与那些满足基于平均值或中值定义的 SLA 的系统相比,这种方法提供了更好的整体体验。

在这篇论文中,有很多关于这个 99.9 % 分布的参考,这反映了亚马逊工程师从客户体验的角度对性能的不懈关注。许多论文报告的是平均数,因此这些都包含在对比较有意义的地方。尽管如此,亚马逊的工程和优化工作并不集中在平均值上。几种技术,例如写入协调器的负载平衡选择,纯粹是针对将性能控制在第 99.9 个百分位。

存储系统通常在建立服务的 SLA 中发挥重要作用,尤其是在业务逻辑相对轻量级的情况下,如许多亚马逊服务的情况。然后状态管理成为服务 SLA 的主要组成部分。 Dynamo 的主要设计考虑因素之一是让服务控制其系统属性,例如持久性和一致性,并让服务在功能、性能和成本效益之间做出自己的权衡。

2.3 设计考虑商业系统中使用的数据复制算法
传统上执行同步复制协调,以提供高度一致的数据访问接口。为了实现这种级别的一致性,这些算法被迫在某些故障场景下权衡数据的可用性。例如,不是处理答案正确性的不确定性,而是在绝对确定它是正确的之前使数据不可用。从很早的复制数据库工作中可以看出,在处理网络故障的可能性时,强一致性和高数据可用性不能同时实现[2, 11]。因此,系统和应用程序需要知道在哪些条件下可以实现哪些特性。

对于容易出现服务器和网络故障的系统,可以通过使用乐观复制技术来提高可用性,其中允许更改在后台传播到副本,并且可以容忍并发、断开连接的工作。这种方法的挑战在于它可能导致必须检测和解决的相互冲突的变化。这种解决冲突的过程引入了两个问题:何时解决以及由谁解决。 Dynamo 旨在成为最终一致的数据存储;也就是说,所有更新最终都会到达所有副本。

一个重要的设计考虑是决定何时执行解决更新冲突的过程,即是否应该在读取或写入期间解决冲突。许多传统的数据存储在写入期间执行冲突解决,并保持读取复杂性简单 [7]。在此类系统中,如果数据存储在给定时间无法到达所有(或大部分)副本,则可能会拒绝写入。另一方面,Dynamo 以“始终可写”的数据存储(即高度可用于写入的数据存储)的设计空间为目标。对于许多亚马逊服务,拒绝客户更新可能会导致糟糕的客户体验。例如,购物车服务必须允许客户在他们的购物车中添加和删除商品,即使在网络和服务器出现故障的情况下也是如此。这一要求迫使我们将冲突解决的复杂性推向读取,以确保写入永远不会被拒绝。

下一个设计选择是谁执行冲突解决过程。这可以由数据存储或应用程序完成。如果冲突解决是由数据存储完成的,那么它的选择是相当有限的。在这种情况下,数据存储只能使用简单的策略,例如“最后写入获胜”[22],来解决冲突更新。另一方面,由于应用程序知道数据模式,它可以决定最适合其客户体验的冲突解决方法。例如,维护客户购物车的应用程序可以选择“合并”冲突的版本并返回一个统一的购物车。尽管有这种灵活性,但一些应用程序开发人员可能不想编写自己的冲突解决机制,而是选择将其下推到数据存储区,而数据存储区又选择了一个简单的策略,例如“最后写入获胜”。

设计中包含的其他关键原则是:

增量可扩展性: Dynamo 应该能够一次横向扩展一个存储主机(以下称为“节点”),对系统运营商和系统本身的影响最小。

对称性: Dynamo 中的每个节点都应该拥有与其对等节点相同的职责集;不应该有一个或多个具有特殊角色或额外职责的节点。根据我们的经验,对称性简化了系统配置和维护的过程。

去中心化:对称性的延伸,设计应该有利于去中心化的点对点技术而不是集中控制。过去,集中控制会导致中断,目标是尽可能避免这种情况。这导致了一个更简单、更可扩展和更可用的系统。

异质性:系统需要能够利用其运行的基础设施中的异质性。例如,工作分配必须与各个服务器的能力成比例。这对于添加具有更高容量的新节点而不必一次升级所有主机至关重要。

3. 相关工作 3.1 点对点系统有几个点对

点(P2P)系统已经研究了数据存储和分发的
问题。第一代 P2P 系统,例如 Freenet 和 Gnutella1

, 是

主要用作文件共享系统。这些是非结构化 P2P 网络的示例,其中对等点之间的覆盖链接是任意建立的。在这些网络中,搜索查询通常通过网络泛滥,以找到尽可能多的共享数据的对等方。 P2P 系统发展到下一代,成为广为人知的结构化 P2P 网络。这些网络采用全球一致的协议来确保任何节点都可以有效地将搜索查询路由到具有所需数据的某个对等点。 Pastry [16] 和 Chord [20] 等系统使用路由机制来确保可以在有限的跳数内回答查询。为了减少多跳路由引入的额外延迟,一些 P2P 系统(例如,[14])采用 O(1) 路由,其中每个对等点在本地维护足够的路由信息,以便它可以将请求(访问数据项)路由到恒定跳数内的适当对等体。

[1] http://freenetproject.org/, http://www.gnutella.org

各种存储系统,例如 Oceanstore [9] 和 PAST [17] 都建立在这些路由覆盖之上。 Oceanstore 提供了一个全局的、事务的、持久的存储服务,支持对广泛复制的数据进行序列化更新。为了允许并发更新,同时避免广域锁定固有的许多问题,它使用基于冲突解决的更新模型。 [21] 中引入了冲突解决以减少事务中止的次数。 Oceanstore 通过处理一系列更新来解决冲突,在其中选择一个总顺序,然后以该顺序原子地应用它们。它是为在不受信任的基础架构上复制数据的环境而构建的。相比之下,PAST 在 Pastry 之上为持久和不可变对象提供了一个简单的抽象层。它假定应用程序可以在其之上构建必要的存储语义(例如可变文件)。

3.2 分布式文件系统和数据库分布式数据的性能、可用性和持久性已在文件系统和数据库系统社区中得到广泛研究。与仅支持平面命名空间的 P2P 存储系统相比,分布式文件系统通常支持分层命名空间。像 Ficus [15] 和 Coda [19] 这样的系统会以牺牲一致性为代价来复制文件以实现高可用性。

通常使用专门的冲突解决程序来管理更新冲突。 Farsite 系统 [1] 是一个分布式文件系统,不使用任何像 NFS 这样的集中式服务器。 Farsite 使用复制实现了高可用性和可扩展性。 Google 文件系统 [6] 是另一个分布式文件系统,用于托管 Google 内部应用程序的状态。 GFS 使用简单的设计,使用单个主服务器来托管整个元数据,并将数据分成块并存储在块服务器中。 Bayou 是一个分布式关系数据库系统,它允许断开连接的操作并提供最终的数据一致性 [21]。

在这些系统中,Bayou、Coda 和 Ficus 允许断开连接的操作,并且对网络分区和中断等问题具有弹性。这些系统的冲突解决程序不同。例如,Coda 和 Ficus 执行系统级冲突解决,而 Bayou 允许应用程序级解决。

然而,所有这些都保证了最终的一致性。与这些系统类似,Dynamo 允许读取和写入操作在网络分区期间继续进行,并使用不同的冲突解决机制解决更新的冲突。

像 FAB [18] 这样的分布式块存储系统将大尺寸对象拆分为更小的块,并以高度可用的方式存储每个块。与这些系统相比,键值存储更适合这种情况,因为:(a)它旨在存储相对较小的对象(大小 < 1M)和(b)键值存储更容易在每个-应用基础。 Antiquity 是一种广域分布式存储系统,旨在处理多个服务器故障 [23]。它使用安全日志来保持数据完整性,在多个服务器上复制每个日志以实现持久性,并使用拜占庭容错协议来确保数据一致性。与 Antiquity 相比,Dynamo 并不关注数据完整性和安全性问题,而是为可信环境而构建的。

Bigtable 是一个用于管理结构化数据的分布式存储系统。它维护一个稀疏的多维排序图,并允许应用程序使用多个属性访问其数据 [2]。与 Bigtable 相比,Dynamo 的目标是只需要键/值访问的应用程序,主要关注高可用性,即使在网络分区或服务器故障之后也不会拒绝更新。

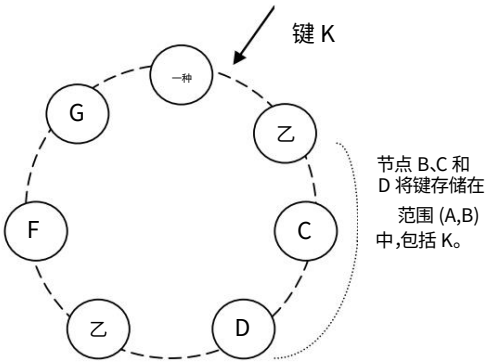

图 2:Dynamo 环中密钥的分区和复制。

传统的复制关系数据库系统关注的是保证复制数据的强一致性问题。

尽管强一致性为应用程序编写者提供了一种方便的编程模型,但这些系统在可扩展性和可用性方面受到限制[7]。这些系统无法处理网络分区,因为它们通常提供强大的一致性保证。

3.3 讨论Dynamo 与上述

分散存储系统的目标需求不同。首先,Dynamo 主要针对需要“始终可写”数据存储的应用程序,其中不会因失败或并发写入而拒绝更新。这是许多亚马逊应用程序的关键要求。其次,如前所述,Dynamo 是为单个管理域内的基础设施而构建的,其中假定所有节点都是可信的。第三,使用 Dynamo 的应用程序不需要支持分层命名空间(许多文件系统中的规范)或复杂的关系模式(传统数据库支持)。第四,Dynamo 专为延迟敏感的应用程序而构建,这些应用程序需要在几百毫秒内执行至少 99.9% 的读写操作。为了满足这些严格的延迟要求,我们必须避免通过多个节点路由请求(这是 Chord 和 Pastry 等几个分布式哈希表系统采用的典型设计)。这是因为多跳路由增加了响应时间的可变性,从而增加了较高百分位数的延迟。 Dynamo 可以被描述为零跳 DHT,其中每个节点在本地维护足够的路由信息以将请求直接路由到适当的节点。

4. 系统架构 需要在生产环境中运行的存储系统架构是

复杂的。除了实际的数据持久化组件外,系统还需要具有可扩展且健壮的解决方案,用于负载平衡、成员资格和故障检测、故障恢复、副本同步、过载处理、状态转移、并发和作业调度、请求编组、请求路由、系统监控报警、配置管理。描述每个解决方案的细节是不可能的,因此本文重点介绍 Dynamo 中使用的核心分布式系统技术:分区、复制、版本控制、成员资格、故障处理和扩展。

表 1: Dynamo中使用的技术总结及其优势。

问题	技术	优势
分区	一致的哈希	增加的 可扩展性
高可用性 用于写入	读取期间具有协调 功能的矢量时钟	版本大小与更新率 分离。
临时处理 失败	马虎的法定人数和 暗示切换	在部分副本不可用 时提供高可用性和持久性保证。
从永久性故障中恢复	使用反熵 默克尔树	在后台同步不同 的副本。
成员资格和故障检测	基于 Gossip 的成 员协议 和故障检测。	保持对称性并避免使用集中式注册表来存储成员资格和节点活跃度信息。

表 1 总结了 Dynamo 使用的技术列表及其各自的优势。

4.1 系统接口Dynamo通过一个简单的

接口来存储一个key关联的对象;它公开了两个操作:get() 和 put()。 get(key) 操作在存储系统中定位与键关联的对象副本,并返回单个对象或具有冲突版本的对象列表以及上下文。 put(key, context, object)操作根据关联的键确定对象的副本应该放置在哪里,并将副本写入磁盘。上下文对关于调用者不透明的对象的系统元数据进行编码,并包括诸如对象版本之类的信息。上下文信息与对象一起存储,以便系统可以验证放置请求中提供的上下文对象的有效性。

Dynamo 将调用者提供的键和对象都视为不透明的字节数组。它在密钥上应用 MD5 哈希以生成 128 位标识符,用于确定负责提供密钥的存储节点。

4.2 分区算法Dynamo 的关键设计要求之一是

它必须增量扩展。这需要一种机制来在系统中的一组节点(即存储主机)上动态划分数据。 Dynamo 的分区方案依赖于一致的散列来将负载分布在多个存储主机上。在一致性散列[10]中,散列函数的输出范围被视为一个固定的循环空间或“环”(即最大的散列值环绕到最小的散列值)。系统中的每个节点在这个空间内被分配一个随机值,代表它在环上的“位置”。通过对数据项的键进行散列以产生其在环上的位置,然后顺时针遍历环以找到位置大于该项位置的第一个节点,将由键标识的每个数据项分配给一个节点。

因此,每个节点都负责环中它与其在环上的前任节点之间的区域。一致散列的主要优点是节点的离开或到达仅影响其直接邻居,而其他节点不受影响。

基本的一致性哈希算法提出了一些挑战。
首先,环上每个节点的随机位置分配导致数据和负载分布不均匀。其次,基本算法忽略了节点性能的异质性。为了解决这些问题,Dynamo 使用了一致性哈希的一种变体(类似于 [10, 20] 中使用的那个):不是将节点映射到圆中的单个点,而是将每个节点分配到环中的多个点。为此,Dynamo 使用了“虚拟节点”的概念。虚拟节点看起来像系统中的单个节点,但每个节点可以负责多个虚拟节点。实际上,当一个新节点被添加到系统中时,它在环中被分配了多个位置(以下称为“令牌”)。微调 Dynamo 的分区方案的过程在第 6 节中讨论。

使用虚拟节点有以下优点:

- 如果某个节点变得不可用(由于故障或日常维护),则该节点处理的负载将均匀地分散在剩余的可用节点上。
- 当一个节点再次可用,或将一个新节点添加到系统中时,新可用节点从每个其他可用节点接受大致等量的负载。
- 一个节点负责的虚拟节点的数量可以根据其容量来决定,考虑到物理基础设施的异构性。

4.3 复制为了实现高可用性和
持久性,Dynamo 将其数据复制到多台主机上。每个数据项在 N 个主机上复制,其中 N 是配置的“每个实例”的参数。每个键k 都分配给一个协调器节点(在上一节中描述)。

协调器负责复制其范围内的数据项。除了本地存储其范围内的每个密钥外,协调器还在环中的 N-1 个顺时针后继节点上复制这些密钥。这导致了一个系统,其中每个节点负责它与其第 N 个前任之间的环区域。在图 2 中,节点 B 复制密钥k

在节点 C 和 D 上,除了将其存储在本地之外。节点 D 将存储在 (A, B], (B, C] 和 (C, D] 范围内的键)。

负责存储特定密钥的节点列表称为首选项列表。该系统的设计将在第 4.8 节中解释,以便系统中的每个节点都可以确定对于任何特定密钥,哪些节点应该在此列表中。为了解决节点故障,首选项列表包含 N 个以上的节点。请注意,通过使用虚拟节点,特定密钥的前 N 个后继位置可能由少于 N 个不同的物理节点拥有(即,一个节点可能拥有前 N 个位置中的一个以上)。为了解决这个问题,一个键的偏好列表是通过跳过环中的位置来构建的,以确保该列表只包含不同的物理节点。

4.4 数据版本控制Dynamo
提供最终一致性,允许更新异步传播到所有副本。 put() 调用可能

在对所有副本应用更新之前返回其调用者,这可能导致后续 get() 操作可能返回没有最新更新的对象的情况。如果没有失败,则存在界限关于更新传播时间。然而,在某些故障情况下(例如,服务器中断或网络分区),更新可能不会在很长一段时间内到达所有副本。

亚马逊平台中有一类应用程序可以容忍这种不一致,并且可以构建为在这些条件下运行。例如,购物车应用程序要求永远不能忘记或拒绝“添加到购物车”操作。如果购物车的最新状态不可用,并且用户对旧版本的购物车进行了更改,则该更改仍然有意义并且应该保留。但与此同时,它不应取代购物车当前不可用的状态,它本身可能包含应保留的更改。请注意, “添加到购物车”和“从购物车中删除商品”操作都被转换为对 Dynamo 的放置请求。当客户想要将商品添加到(或从)购物车中删除并且最新版本不可用时,该商品将添加到(或从)旧版本中删除,并且稍后会协调不同的版本。

为了提供这种保证,Dynamo 将每次修改的结果视为新的、不可变的版本

数据。它允许对象的多个版本同时存在于系统中。大多数时候,新版本包含以前的版本,系统本身可以确定权威版本(句法协调)。

但是,版本分支可能会发生,在故障与并发更新相结合的情况下,导致对象的版本冲突。在这些情况下,系统无法协调同一对象的多个版本,客户端必须执行协调才能将数据演化的多个分支合并为一个(语义协调)。折叠操作的一个典型示例是“合并”客户购物车的不同版本。使用这种协调机制,“添加到购物车”操作永远不会丢失。但是,已删除的项目可能会重新出现。

重要的是要了解某些故障模式可能导致系统不仅具有两个版本,而且具有多个版本的相同数据。存在网络分区和节点故障时的更新可能会导致对象具有不同的版本子历史记录,系统将来需要对其进行协调。这要求我们设计的应用程序明确承认同一数据的多个版本的可能性(以便永远不会丢失任何更新)。

Dynamo 使用矢量时钟 [12] 来捕捉同一对象的不同版本之间的因果关系。矢量时钟实际上是(节点,计数器)对的列表。一个矢量时钟与每个对象的每个版本相关联。通过检查它们的矢量时钟,可以确定一个对象的两个版本是否在并行分支上或具有因果顺序。如果第一个对象时钟上的计数器小于或等于第二个时钟中的所有节点,则第一个是第二个的祖先,可以被遗忘。否则,这两个变化被认为是冲突的,需要和解。

在 Dynamo 中,当客户端希望更新对象时,它必须指定要更新的版本。这是通过传递从较早的读取操作中获得的上下文来完成的,其中包含矢量时钟信息。在处理读取请求时,如果

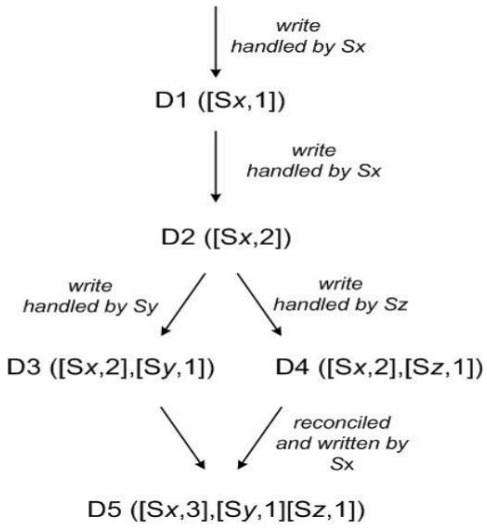

图 3:对象随时间的版本演变。

Dynamo 可以访问多个无法在语法上协调的分支,它将返回叶子处的所有对象,以及上下文中的相应版本信息。使用此上下文的更新被认为已经协调了不同的版本,并且分支被折叠成一个新版本。

为了说明矢量时钟的使用,让我们考虑图 3 中所示的示例。客户端写入一个新对象。处理此键写入的节点(例如 Sx)增加其序列号并使用它来创建数据的矢量时钟。系统现在具有对象 D1 及其关联的时钟 [(Sx, 1)]。客户端更新对象。假设同一个节点也处理这个请求。系统现在还具有对象 D2 及其关联的时钟 [(Sx, 2)]。 D2 从 D1下降,因此会覆盖 D1,但是 D1 的副本可能会在尚未看到 D2 的节点上徘徊。让我们假设同一个客户端再次更新对象并且不同的服务器(比如 Sy)处理请求。系统现在有数据 D3 及其相关的时钟 [(Sx, 2), (Sy, 1)]。

接下来假设另一个客户端读取 D2,然后尝试更新它,另一个节点(比如 Sz)进行写入。系统现在有 D4(D2 的后代),其版本时钟为 [(Sx, 2), (Sz, 1)]。知道 D1 或 D2 的节点可以在接收到 D4 及其时钟后确定 D1 和 D2 被新数据覆盖并且可以被垃圾收集。一个知道 D3 并接收到 D4 的节点会发现它们之间没有因果关系。换言之,D3 和 D4 中存在相互不反映的变化。两个版本的数据都必须保留并呈现给客户端(在读取时)以进行语义协调。

现在假设某个客户端同时读取了 D3 和 D4(上下文将反映这两个值都是由读取找到的)。读取的上下文是对D3和D4的时钟的总结,即[(Sx, 2), (Sy, 1), (Sz, 1)]。如果客户端执行协调并且节点 Sx 协调写入,则 Sx 将在时钟中更新其序列号。新数据 D5 将具有以下时钟:[(Sx, 3), (Sy, 1), (Sz, 1)]。

矢量时钟的一个可能问题是,如果许多服务器协调写入一个

目的。实际上,这不太可能,因为写入通常由首选项列表中的前 N 个节点之一处理。在网络分区或多个服务器故障的情况下,写入请求可能由不在首选项列表中的前 N 个节点中的节点处理,从而导致矢量时钟的大小增加。在这些情况下,希望限制矢量时钟的大小。为此,Dynamo 采用以下时钟截断方案:与每个(节点、计数器)对一起,Dynamo 存储一个时间戳,指示节点最后一次更新数据项的时间。当向量时钟中的(节点,计数器)对的数量达到阈值(比如 10)时,从时钟中删除最旧的对。

显然,这种截断方案会导致协调效率低下,因为无法准确地导出后代关系。然而,这个问题并没有在生产中浮出水面,因此这个问题还没有得到彻底的调查。

4.5 get() 和 put() 操作的执行Dynamo 中的任何存储节点都有资格接收客户端对任何 key 的 get 和 put 操作。在本节中,为简单起见,我们将描述如何在无故障环境中执行这些操作,在后续部分中,我们将描述在故障期间如何执行读写操作。

get 和 put 操作都是使用 Amazon 的特定于基础设施的请求处理框架通过 HTTP 调用的。
客户端可以使用两种策略来选择节点:(1)通过通用负载均衡器路由其请求,该负载均衡器将根据负载信息选择节点,或(2)使用分区感知客户端库直接路由请求到适当的协调节点。第一种方法的优点是客户端不必在其应用程序中链接任何特定于 Dynamo 的代码,而第二种策略可以实现更低的延迟,因为它跳过了潜在的转发步骤。

处理读取或写入操作的节点称为协调器。通常,这是首选项列表中前 N 个节点中的第一个。如果请求是通过负载平衡器接收的,则访问密钥的请求可能会被路由到环中的任何随机节点。在这种情况下,如果接收到请求的节点不在请求键的首选项列表的前 N 中,则该节点将不会协调它。相反,该节点会将请求转发到首选项列表中前 N 个节点中的第一个。

读写操作涉及首选项列表中的前 N 个健康节点,跳过那些关闭或无法访问的节点。
当所有节点都健康时,访问密钥首选项列表中的前 N 个节点。当存在节点故障或网络分区时,将访问在首选项列表中排名较低的节点。

为了保持其副本之间的一致性,Dynamo 使用类似于仲裁系统中使用的一致性协议。
该协议有两个关键的可配置值:R 和 W。R 是必须参与成功读取操作的最小节点数。 W 是必须参与成功写入操作的最小节点数。设置 R 和 W 使得 R + W > N 产生一个类似群体的系统。在此模型中,get(或 put)操作的延迟由最慢的 R(或 W)副本决定。因此,R 和 W 通常配置为小于 N,以提供更好的延迟。

在收到对密钥的 put() 请求后,协调器为新版本生成矢量时钟并在本地写入新版本。然后协调器发送新版本(连同

新的向量时钟)到 N 个最高级别的可达节点。如果至少有 W-1 个节点响应,则认为写入成功。

类似地,对于 get() 请求,协调器从该键的首选项列表中排名最高的 N 个可达节点请求该键的所有现有数据版本,然后等待 R 个响应,然后将结果返回给客户端。如果协调器最终收集了多个版本的数据,它会返回它认为不相关的所有版本。然后协调不同的版本,并写回取代当前版本的协调版本。

4.6 处理故障:提示切换如果 Dynamo 使用传统的仲裁方法,它将在服务器故障和网络分区期间不可用,并且即使在最简单的故障条件下也会降低持久性。为了解决这个问题,它没有强制执行严格的法定人数,而是使用“草率的法定人数”;所有读取和写入操作都在首选项列表中的前 N个健康节点上执行,这可能并不总是在遍历一致哈希环时遇到的前 N 个节点。

考虑图 2 中给出的 N=3 的 Dynamo 配置示例。在此示例中,如果节点 A 在写入操作期间暂时关闭或无法访问,则通常位于 A 上的副本现在将被发送到节点 D。这样做是为了保持所需的可用性和持久性保证。发送到 D 的副本将在其元数据中包含一个提示,表明哪个节点是副本的预期接收者(在本例中为 A)。接收到提示副本的节点会将它们保存在一个单独的本地数据库中,该数据库会定期扫描。在检测到 A 已经恢复后,D 将尝试将副本交付给 A。一旦传输成功,D 可以从其本地存储中删除对象,而不会减少系统中的副本总数。

使用提示切换,Dynamo 确保读写操作不会由于临时节点或网络故障而失败。需要最高可用性级别的应用程序可以将 W 设置为 1,这确保只要系统中的单个节点已将密钥持久地写入其本地存储,就可以接受写入。因此,只有当系统中的所有节点都不可用时,写入请求才会被拒绝。然而,在实践中,生产中的大多数亚马逊服务都设置了更高的 W 以满足所需的持久性水平。第 6 节对配置 N、R 和 W 进行了更详细的讨论。

高度可用的存储系统必须能够处理整个数据中心的故障。数据中心故障的发生是由于停电、冷却故障、网络故障和自然灾害。 Dynamo 的配置使得每个对象都可以跨多个数据中心进行复制。本质上,密钥的偏好列表是这样构建的,使得存储节点分布在多个数据中心。这些数据中心通过高速网络链路连接。这种跨多个数据中心复制的方案使我们能够处理整个数据中心的故障而不会出现数据中断。

4.7 处理永久性故障:副本同步如果系统成员流失率低且节点故障是暂时的,则提示切换效果最好。在某些情况下,提示副本在返回之前变得不可用

原始副本节点。为了处理这个和其他对持久性的威胁,Dynamo 实现了一个反熵(副本同步)协议来保持副本同步。

为了更快地检测副本之间的不一致并最小化传输的数据量,Dynamo 使用 Merkle 树 [13]。 Merkle 树是一个哈希树,其中叶子是单个键值的哈希值。树中较高的父节点是它们各自子节点的哈希值。 Merkle 树的主要优点是可以独立检查树的每个分支,而不需要节点下载整个树或整个数据集。此外,Merkle 树有助于减少需要传输的数据量,同时检查副本之间的不一致。例如,如果两棵树的根的哈希值相等,则树中的叶子节点的值相等,节点不需要同步。如果不是,则意味着某些副本的值不同。在这种情况下,节点可能会交换子节点的哈希值,并且该过程继续进行,直到它到达树的叶子,此时主机可以识别“不同步”的密钥。 Merkle 树最大限度地减少了需要传输的数据量

同步并减少反熵过程中执行的磁盘读取次数。

Dynamo 使用 Merkle 树进行反熵,如下所示:每个节点为其托管的每个键范围(虚拟节点覆盖的键集)维护一个单独的 Merkle 树。这允许节点比较键范围内的键是否是最新的。在该方案中,两个节点交换与它们共同托管的键范围相对应的 Merkle 树的根。

随后,使用上述树遍历方案,节点确定它们是否有任何差异并执行适当的同步操作。这种方案的缺点是,当节点加入或离开系统时,许多键范围会发生变化,从而需要重新计算树。

然而,这个问题可以通过第 6.2 节中描述的细化分区方案来解决。

4.8 成员资格和故障检测4.8.1 环成员资格在 Amazon 的环境中,节点中断(由于故障和维护任务)通常是暂时的,但可能会持续较长时间。节点中断很少表示永久离开,因此不应导致重新平衡分区分配或修复无法访问的副本。同样,手动错误可能会导致新 Dynamo 节点的意外启动。由于这些原因,使用显式机制来启动从 Dynamo 环中添加和删除节点被认为是合适的。管理员使用命令行工具或浏览器连接到 Dynamo 节点并发出成员资格更改以将节点加入环或从环中删除节点。为请求提供服务的节点将成员资格更改及其发布时间写入持久存储。成员资格更改形成历史记录,因为可以多次删除和添加节点。基于 gossip 的协议传播成员变化并保持成员的最终一致视图。每个节点每秒都会联系一个随机选择的对等节点,这两个节点有效地协调它们持久的成员资格更改历史。

当一个节点第一次启动时,它会选择它的令牌集(一致哈希空间中的虚拟节点)并将节点映射到它们各自的令牌集。映射保存在磁盘上,并且

最初只包含本地节点和令牌集。存储在不同 Dynamo 节点的映射在协调成员更改历史的同一通信交换期间协调。因此,分区和放置信息也通过基于 gossip 的协议传播,并且每个存储节点都知道其对等点处理的令牌范围。这允许每个节点将密钥的读/写操作直接转发到正确的节点集。

4.8.2 外部发现上述机制可能会暂时导致逻辑分区的 Dynamo 环。

例如,管理员可以联系节点 A 将 A 加入环,然后联系节点 B 将 B 加入环。在这种情况下,节点 A 和 B 将各自认为自己是环的成员,但两者都不会立即意识到对方。为了防止逻辑分区,一些 Dynamo 节点扮演种子的角色。种子是通过外部机制发现并为所有节点所知的节点。因为所有节点最终都会通过种子来协调它们的成员资格,所以逻辑分区的可能性很小。

种子可以从静态配置或配置服务中获得。通常,种子是 Dynamo 环中功能齐全的节点。

4.8.3 故障检测Dynamo 中的故障检

测用于避免在 get() 和 put() 操作期间以及在传输分区和提示副本时尝试与无法访问的对等方通信。

为了避免失败的通信尝试,纯本地的故障检测概念是完全足够的:如果节点 B 不响应节点 A 的消息(即使 B 响应节点 C 的消息),节点 A 可能认为节点 B 失败。在 Dynamo 环中存在稳定速率的客户端请求产生节点间通信的情况下,当 B 未能响应消息时,节点 A 快速发现节点 B 没有响应;然后节点 A 使用备用节点为映射到 B 分区的请求提供服务; A 定期重试 B 以检查后者的恢复情况。在没有客户端请求驱动两个节点之间的流量的情况下,任何一个节点都不需要知道另一个节点是否可以访问和响应。

分散式故障检测协议使用简单的八卦式协议,使系统中的每个节点都能够了解其他节点的到达(或离开)。有关分散式故障检测器和影响其准确性的参数的详细信息,有兴趣的读者可以参考 [8]。 Dynamo 的早期设计使用分散式故障检测器来维护全局一致的故障状态视图。后来确定显式节点加入和离开方法消除了对故障状态的全局视图的需要。这是因为通过显式节点加入和离开方法通知节点永久节点添加和删除,并且当单个节点无法与其他节点通信时(在转发请求时)检测到临时节点故障。

4.9 添加/删除存储节点当一个新节点(比如 X)被添加到系统中时,它会被分配一些随机分散在环上的令牌。对于分配给节点 X 的每个键范围,可能有许多节点(小于或等于 N)当前负责处理属于其令牌范围内的键。由于将键范围分配给 X,一些现有节点不再需要它们的某些键,这些节点将这些键转移给 X。让

我们考虑一个简单的引导场景,其中节点 X 被添加到图 2 所示的 A 和 B 之间的环中。当 X 被添加到系统时,它负责在 (F, G], (G, A] 和 (A, X]。因此,节点 B、C 和 D 不再需要将密钥存储在这些各自的范围内。

因此,节点 B、C 和 D 将向 X 提供并在确认后转移适当的密钥集。当一个节点从系统中移除时,密钥的重新分配发生在一个相反的过程中。

操作经验表明,这种方法将密钥分发的负载均匀地分布在存储节点上,这对于满足延迟要求和确保快速引导非常重要。最后,通过在源和目标之间添加确认轮次,可以确保目标节点不会收到给定密钥范围的任何重复传输。

5.实现 在 Dynamo 中,每个存储节点都具

有三个主要的软件组件:请求协调、成员资格和故障检测以及本地持久性引擎。所有这些组件都是用 Java 实现的。

Dynamo 的本地持久性组件允许插入不同的存储引擎。正在使用的引擎是 Berkeley Database (BDB) Transactional Data Store2

, BDB Java 版、MySQL 和具有持久后备存储的内存缓冲区。设计可插拔持久性组件的主要原因是选择最适合应用程序访问模式的存储引擎。例如,BDB 通常可以处理数十千字节的对象,而 MySQL 可以处理更大大小的对象。应用程序根据其对象大小分布选择 Dynamo 的本地持久性引擎。

Dynamo 的大多数生产实例都使用 BDB Transactional Data Store。

请求协调组件构建在事件驱动的消息传递基板之上,其中消息处理管道分为多个阶段,类似于 SEDA 架构 [24]。

所有通信都使用 Java NIO 通道实现。
协调器通过从一个或多个节点收集数据(在读取的情况下)或将数据存储在一个或多个节点(用于写入)来代表客户端执行读取和写入请求。每个客户端请求都会导致在接收客户端请求的节点上创建一个状态机。状态机包含用于识别负责密钥的节点、发送请求、等待响应、可能进行重试、处理响应并将响应打包到客户端的所有逻辑。

每个状态机实例只处理一个客户端请求。
例如,读取操作实现以下状态机:(i)向节点发送读取请求,(ii)等待所需响应的最小数量,(iii)如果在给定的时间范围内收到的回复太少,则失败请求,(iv) 否则收集所有数据版本并确定要返回的版本,以及 (v) 如果启用了版本控制,则执行语法协调并生成包含包含所有剩余版本的向量时钟的不透明写入上下文。为简洁起见,省略了故障处理和重试状态。

读取响应返回给调用者后,状态

[2] http://www.oracle.com/database/berkeley-db.html

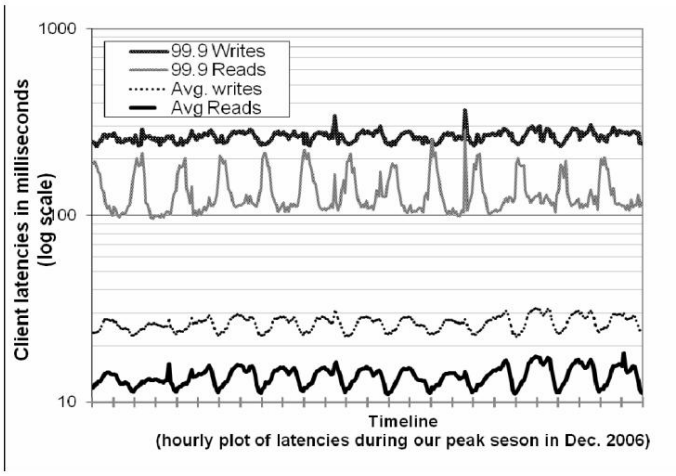

图 4:在 2006 年 12 月的请求高峰期,读写请求的平均延迟和 99.9% 的延迟。

x 轴上连续刻度之间的间隔对应于 12 小时。延迟遵循类似于请求率的昼夜模式,99.9% 的延迟比平均值高一个数量级

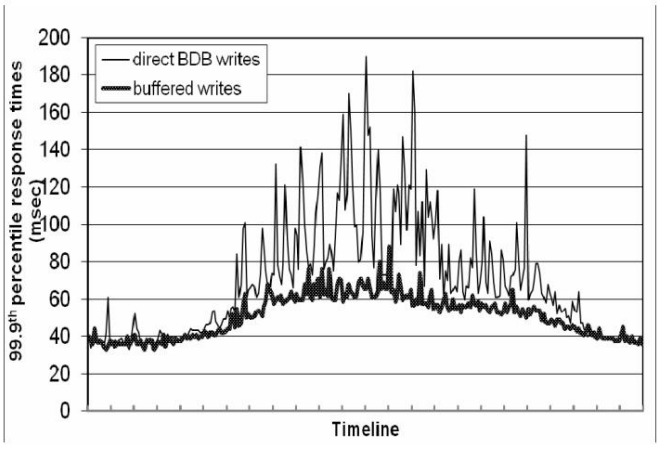

图 5:缓冲写入与非缓冲写入在 24 小时内的 99.9% 延迟性能比较。 x 轴上连续刻度之间的间隔

对应一小时。

机器等待一小段时间来接收任何未完成的响应。如果在任何响应中返回过时版本,协调器将使用最新版本更新这些节点。此过程称为读取修复,因为它会在机会主义的时间修复错过最近更新的副本,并使反熵协议不必这样做。

如前所述,写入请求由首选项列表中的前 N 个节点之一协调。尽管总是希望前 N 个节点中的第一个节点来协调写入,从而将所有写入序列化在一个位置,但这种方法会导致负载分布不均匀,从而导致违反 SLA。这是因为请求负载不是在对象之间均匀分布的。为了解决这个问题,允许首选项列表中的前 N 个节点中的任何一个来协调写入。特别是,由于每次写入通常都在读取操作之后,因此写入的协调器被选择为对存储在请求的上下文信息中的先前读取操作响应最快的节点。这种优化使我们能够选择具有由先前读取操作读取的数据的节点,从而增加获得“read-your-writes”一致性的机会。

它还减少了请求处理性能的可变性,从而提高了 99.9 个百分位的性能。

6. 经验和教训Dynamo 被多个具有不同配置的服务所使用。

这些实例的不同之处在于它们的版本协调逻辑和读/写仲裁特征。以下是使用 Dynamo 的主要模式:

- 业务逻辑特定的协调:这是 Dynamo 的一个流行用例。每个数据对象都跨多个节点复制。在不同版本的情况下,客户端应用程序执行自己的协调逻辑。前面讨论的购物车服务就是这个类别的一个典型例子。它的业务逻辑通过合并客户购物车的不同版本来协调对象。
- 基于时间戳的协调:这种情况与前一种情况的不同之处仅在于协调机制。在不同版本的情况下,Dynamo 执行简单的基于时间戳的“最后写入获胜”的协调逻辑;即,选择具有最大物理时间戳值的对象作为正确版本。维护客户会话信息的服务就是使用这种模式的服务的一个很好的例子。
- 高性能读取引擎:虽然 Dynamo 被构建为“始终可写”的数据存储,但一些服务正在调整其仲裁特性并将其用作高性能读取引擎。通常,这些服务具有很高的读取请求率并且只有少量更新。在此配置中,通常 R 设置为 1,W 设置为 N。对于这些服务,Dynamo 提供了跨多个节点分区和复制其数据的能力,从而提供增量可扩展性。其中一些实例用作存储在更重量级后备存储中的数据的权威持久性缓存。维护产品目录和促销品的服务属于此类别。

Dynamo 的主要优势在于其客户端应用程序可以调整 N、R 和 W 的值,以实现其所需的性能、可用性和耐用性水平。例如,N 的值决定了每个对象的持久性。Dynamo 的用户使用的 N 的典型值为 3。

W 和 R 的值影响对象的可用性、持久性和一致性。例如,如果 W 设置为 1,那么只要系统中至少有一个节点可以成功处理写入请求,系统就永远不会拒绝写入请求。但是,W 和 R 的低值会增加不一致的风险,因为写入请求被视为成功并返回给客户端,即使它们没有被大多数副本处理。当写入请求成功返回到客户端时,即使它仅在少数节点上持久化,这也会引入持久性漏洞窗口。

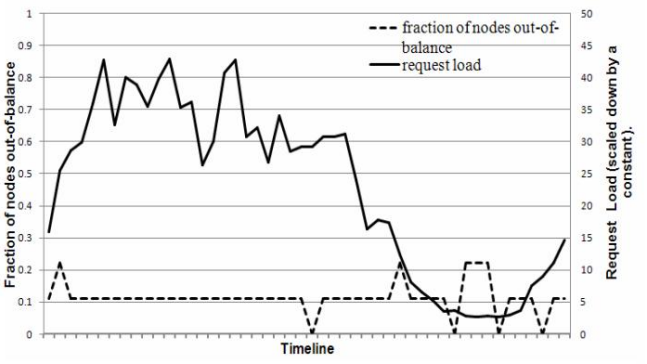

图 6:不平衡节点的比例（即请求负载高于某个阈值的节点）

平均系统负载)及其相应的请求负载。
x 轴刻度之间的间隔对应于时间
30分钟的时间。

传统智慧认为,耐用性和可用性齐头并进。但是,这里不一定是这样。例如,可以通过增加 W 来减少持久性的漏洞窗口。这可能会增加拒绝请求的可能性（从而降低可用性）,因为需要更多的存储主机处于活动状态才能处理写入请求。

几个 Dynamo 实例使用的常见 (N,R,W) 配置是 (3,2,2)。选择这些值是为了满足性能、持久性、一致性和可用性 SLA 的必要级别。

本节中介绍的所有测量都是在使用 (3,2,2) 配置运行并运行具有同质硬件配置的几百个节点的实时系统上进行的。如前所述,Dynamo 的每个实例都包含位于多个数据中心的节点。这些数据中心通常通过高速网络链路连接。回想一下,要生成成功的 get（或 put)响应,R（或 W)节点需要响应协调器。显然,数据中心之间的网络延迟会影响响应时间,并且选择节点（及其数据中心位置)以满足应用程序的目标 SLA。

6.1 平衡性能和耐用性 虽然 Dynamo 的主要设计目标是构建一个高度可用的数据存储,但性能是亚马逊平台中同样重要的标准。如前所述,为了提供一致的客户体验,亚马逊的服务将其绩效目标设定在更高的百分位（例如99.9th 或 99.99th

百分位数）。使用 Dynamo 的服务所需的典型 SLA 是 99.9% 的读写请求在 300 毫秒内执行。

由于 Dynamo 在 I/O 吞吐量远低于高端企业服务器的标准商品硬件组件上运行,因此为读写操作提供始终如一的高性能是一项艰巨的任务。多个存储节点参与读取和写入操作使其更具挑战性,因为这些操作的性能受到最慢的 R 或 W 副本的限制。图 4 显示了Dynamo 读取和写入操作在 30 天内的平均延迟和 99.9 % 的延迟。如图所示,延迟表现出明显的昼夜模式,这是传入请求率中昼夜模式的结果（即,有一个

白天和晚上的请求率存在显着差异）。此外,写入延迟明显高于读取延迟,因为写入操作总是会导致磁盘访问。

此外,第 99.9个百分位数的延迟约为 200 毫秒,比平均值高一个数量级。这是因为 99.9% 的延迟受多个因素的影响,例如请求负载的可变性、对象大小和位置模式。

虽然这种性能水平对于许多服务来说是可以接受的,但一些面向客户的服务需要更高水平的性能。对于这些服务,Dynamo 提供了权衡耐久性保证以换取性能的能力。在优化中,每个存储节点在其主存中维护一个对象缓冲区。每个写入操作都存储在缓冲区中,并由写入线程定期写入存储。在这个方案中,读取操作首先检查请求的键是否存在于缓冲区中。如果是这样,则从缓冲区而不是存储引擎中读取对象。

这种优化导致在高峰流量期间将 99.9% 的延迟降低了 5 倍,即使对于一千个对象的非常小的缓冲区也是如此（参见图 5）。此外,如图所示,写入缓冲可以消除较高的百分位延迟。

显然,该方案以耐久性换取性能。在此方案中,服务器崩溃可能会导致丢失在缓冲区中排队的写入。为了降低持久性风险,写入操作被改进为让协调器从 N 个副本中选择一个来执行“持久写入”。由于协调器只等待W个响应,因此写入操作的性能不受单个副本执行的持久写入操作的性能影响。

6.2 确保均匀的负载分布Dynamo 使用一致的散列在其副本之间划分其密钥空间并确保均匀的负载分布。假设密钥的访问分布没有高度倾斜,统一的密钥分布可以帮助我们实现均匀的负载分布。特别是,Dynamo 的设计假设即使在访问分布存在明显偏差的地方,分布的流行端也有足够的密钥,因此处理流行密钥的负载可以通过分区均匀地分布在节点上。本节讨论在 Dynamo 中看到的负载不平衡以及不同分区策略对负载分布的影响。

为了研究负载不平衡及其与请求负载的相关性,每个节点接收的请求总数在 24 小时内被测量 - 细分为 30 分钟的间隔。在给定的时间窗口中,如果节点的请求负载偏离平均负载的值 a 小于某个阈值（此处为 15%）,则认为该节点处于“平衡”状态。否则该节点被视为“失衡”。图 6 显示了在此时间段内“失衡”（以下简称“失衡率”)的节点比例。作为参考,还绘制了整个系统在此时间段内收到的相应请求负载。如图所示,不平衡率随着负载的增加而降低。例如,在低负载时,不平衡率高达 20%,而在高负载时,则接近 10%。直观地说,这可以通过以下事实来解释:在高负载下,会访问大量流行的键,并且由于键的均匀分布,负载是均匀分布的。然而,在低负载期间（负载为 1/8

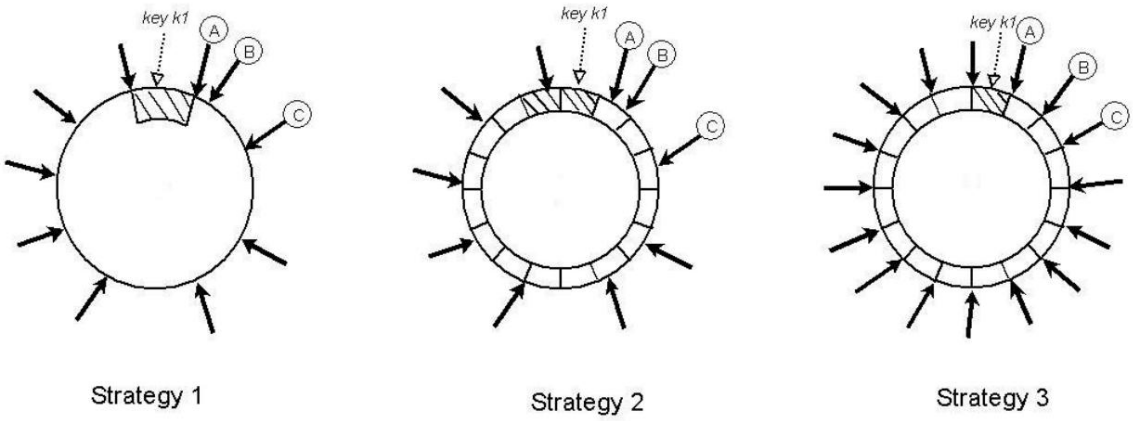

图 7:三种策略中键的分区和放置。 A、B 和 C 描述了三个唯一节点,它们构成了一致哈希环 (N=3) 上密钥 k1 的偏好列表。阴影区域表示节点 A 的键范围,

B 和 C 形成偏好列表。暗箭头表示各个节点的令牌位置。

测量的峰值负载),访问的流行键较少,导致负载不平衡度更高。

本节讨论 Dynamo 的分区方案如何随着时间的推移而演变及其对负载分布的影响。

策略 1:每个节点有 T 个随机令牌并按令牌值分区:这是在生产中部署的初始策略(并在第 4.2 节中描述)。在这个方案中,每个节点都被分配了 T 个令牌(从哈希空间中均匀随机选择)。所有节点的令牌根据它们在哈希空间中的值进行排序。每两个连续的标记定义一个范围。最后一个标记和第一个标记形成一个范围,该范围从哈希空间中的最大值“环绕”到最小值。因为令牌是随机选择的,所以范围的大小会有所不同。随着节点加入和离开系统,令牌集会发生变化,因此范围也会发生变化。请注意,维护每个节点的成员资格所需的空间随着系统中节点的数量线性增加。

在使用该策略时,遇到了以下问题。首先,当一个新节点加入系统时,它需要从其他节点“窃取”其密钥范围。但是,将键范围移交给新节点的节点必须扫描其本地持久性存储以检索适当的数据项集。

请注意,在生产节点上执行这样的扫描操作很棘手,因为扫描是高度资源密集型操作,它们需要在后台执行而不影响客户性能。这要求我们以最低优先级运行引导任务。但是,这显着减慢了引导过程,并且在繁忙的购物季节,当节点每天处理数百万个请求时,引导过程几乎需要一天时间才能完成。其次,当一个节点加入/离开系统时,许多节点处理的键范围发生变化,需要重新计算新范围的默克尔树,这是在生产系统上执行的一项重要操作。最后,由于密钥范围的随机性,没有简单的方法对整个密钥空间进行快照,这使得归档过程变得复杂。在这种方案中,归档整个密钥空间需要我们分别从每个节点检索密钥,这是非常低效的。

这种策略的根本问题是数据分区和数据放置的方案是相互交织的。例如,在某些情况下,最好向系统添加更多节点以处理请求负载的增加。但是,在这种情况下,不可能在不影响数据分区的情况下添加节点。理想情况下,最好使用独立的方案进行分区和放置。为此,评估了以下策略:

策略 2:每个节点有 T 个随机令牌和大小相等的分区:在这个策略中,哈希空间被划分为 Q 个大小相等的分区/范围,每个节点都分配有 T 个随机令牌。 Q 通常设置为 Q >> N 和 Q >> S*T,其中 S 是系统中的节点数。在此策略中,令牌仅用于构建将哈希空间中的值映射到节点的有序列表的函数,而不是决定分区。在从分区末端顺时针遍历一致哈希环时遇到的前 N 个唯一节点上放置一个分区。图 7 说明了 N=3 的这种策略。在此示例中,节点 A、B、C 在从包含密钥 k1 的分区的末尾开始遍历环时遇到。该策略的主要优点是:(i)分区和分区放置的解耦,以及(ii)能够在运行时更改放置方案。

策略 3:每个节点 Q/S 个令牌,大小相等的分区:与策略 2 类似,该策略将哈希空间划分为 Q 个大小相等的分区,并且分区的放置与分区方案解耦。此外,每个节点都分配有 Q/S 令牌,其中 S 是系统中的节点数。当一个节点离开系统时,它的令牌会随机分配给剩余的节点,从而保留这些属性。

类似地,当一个节点加入系统时,它会以一种保留这些属性的方式从系统中的节点“窃取”令牌。

针对 S=30 和 N=3 的系统评估这三种策略的效率。然而,以公平的方式比较这些不同的策略是很困难的,因为不同的策略有不同的配置来调整它们的效率。例如,策略 1 的负载分布属性取决于令牌的数量(即 T),而策略 3 取决于分区的数量(即 Q)。比较这些策略的一种公平方法是

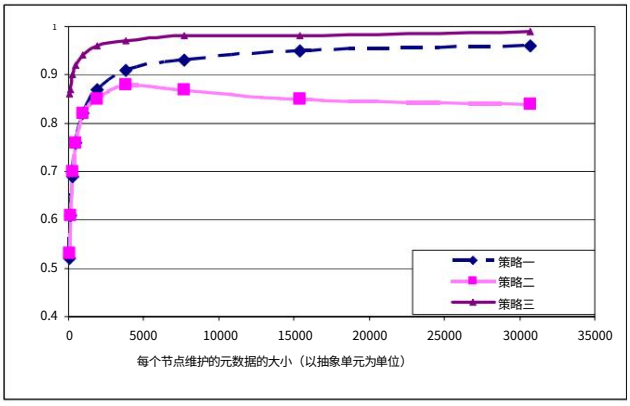

图8:负载分配效率对比
具有 30 个节点和 N=3 的系统的不同策略
在每个节点上维护等量的元数据。这
系统大小和副本数的值基于
为我们大多数人部署的典型配置
服务。

评估其负载分布中的偏差,而所有策略都使用相同数量的空间来维护其成员信息。例如,在策略 1 中,每个节点都需要维护环中所有节点的令牌位置,而在策略 3 中,每个节点都需要维护有关分配给每个节点的分区的信息。

在我们的下一个实验中,通过改变相关参数（T 和 Q)来评估这些策略。每个策略的负载均衡效率是针对每个节点需要维护的不同大小的成员信息来衡量的,其中负载均衡效率定义为每个节点服务的平均请求数与最大服务请求数的比值。最热的节点。

结果如图 8 所示。从图中可以看出,策略 3 的负载均衡效率最高,策略 2 的负载均衡效率最差。在将 Dynamo 实例从使用策略 1 迁移到策略 3 的过程中,策略 2 短暂地充当了临时设置。与策略 1 相比,策略 3 实现了更高的效率,并将每个节点维护的成员信息大小减少了三个数量级。虽然存储不是主要问题,但节点会定期发送成员信息,因此希望尽可能保持此信息紧凑。除此之外,出于以下原因,策略 3 具有优势且易于部署： (i)更快的引导/恢复：

由于分区范围是固定的,因此它们可以存储在单独的文件中,这意味着可以通过简单地传输文件将分区作为一个单元重新定位（避免定位特定项目所需的随机访问）。这简化了引导和恢复的过程。 (ii)易于归档:数据集的定期归档是大多数亚马逊存储服务的强制性要求。

归档 Dynamo 存储的整个数据集在策略 3 中更简单,因为分区文件可以单独归档。

相比之下,在策略 1 中,令牌是随机选择的,归档存储在 Dynamo 中的数据需要分别从各个节点检索密钥,通常效率低下且速度慢。策略 3 的缺点是更改节点成员资格需要协调以保持分配所需的属性。

6.3 不同的版本:何时以及有多少?

如前所述,Dynamo 旨在权衡可用性的一致性。要了解不同故障对一致性的精确影响,需要有关多个因素的详细数据:停机时间、故障类型、组件可靠性、工作负载等。

详细介绍这些数字超出了本文的范围。但是,本节讨论了一个很好的总结指标:应用程序在实时生产环境中看到的不同版本的数量。

数据项的不同版本出现在两种情况下。首先是当系统面临节点故障、数据中心故障、网络分区等故障场景时。第二个是当系统处理单个数据项的大量并发写入者并且多个节点最终同时协调更新时。从可用性和效率的角度来看,最好在任何给定时间将不同版本的数量保持在尽可能低的水平。如果版本不能仅基于矢量时钟在语法上进行协调,则必须将它们传递给业务逻辑以进行语义协调。语义协调在服务上引入了额外的负载,因此希望最大限度地减少对它的需求。

在我们的下一个实验中,返回到购物车服务的版本数量被分析了 24 小时。

在此期间,99.94% 的请求只看到一个版本； 0.00057% 的请求看到了 2 个版本； 0.00047% 的请求看到 3 个版本,0.00009% 的请求看到 4 个版本。这表明很少创建不同的版本。

经验表明,不同版本数量的增加不是由于失败,而是由于并发写入者数量的增加。并发写入数量的增加通常由繁忙的机器人（自动化客户端程序)触发,很少由人类触发。由于故事的敏感性,这个问题没有详细讨论。

6.4 客户端驱动或服务器驱动的协调如第 5 节所述,Dynamo 有一个请求协调组件,它使用状态机来处理传入的请求。

客户端请求由负载均衡器统一分配给环中的节点。任何 Dynamo 节点都可以充当读取请求的协调者。另一方面,写入请求将由密钥当前首选项列表中的节点协调。这个限制是因为这些首选节点有额外的责任来创建一个新的版本标记,该标记会包含已被写入请求更新的版本。

请注意,如果 Dynamo 的版本控制方案基于物理时间戳,则任何节点都可以协调写入请求。

请求协调的另一种方法是将状态机移动到客户端节点。在此方案中,客户端应用程序使用库在本地执行请求协调。

客户端定期选择一个随机 Dynamo 节点并下载其当前的 Dynamo 成员状态视图。使用此信息,客户端可以确定哪一组节点形成任何给定密钥的首选项列表。读取请求可以在客户端节点进行协调,从而避免负载均衡器将请求分配给随机 Dynamo 节点时产生的额外网络跳跃。写入将被转发到密钥首选项列表中的节点,或者可以

表 2:客户端驱动和服务器驱动的协调方法的性能。

	99.9% 读取延迟（毫秒）	99.9% 写入延迟（毫秒）	平均读取延迟（毫秒）	平均写入延迟（毫秒）
服务器驱动	68.9	68.5	3.9	4.02
客户驱动	30.4	30.4	1.55	1.9

如果 Dynamo 使用基于时间戳的版本控制,则在本地进行协调。

客户端驱动的协调方法的一个重要优点是不再需要负载均衡器来均匀分布客户端负载。通过向存储节点近乎均匀地分配密钥,可以隐含地保证公平的负载分布。显然,该方案的效率取决于会员信息在客户端的新鲜程度。目前,客户端每 10 秒轮询一次随机 Dynamo 节点以获取成员更新。选择基于拉的方法而不是基于推送的方法,因为前者可以更好地扩展大量客户端,并且需要在服务器上维护与客户端有关的非常少的状态。但是,在最坏的情况下,客户端可能会在 10 秒内暴露于陈旧的成员资格。如果客户端检测到它的成员表是陈旧的（例如,当某些成员不可达时）,它会立即刷新它的成员信息。

表 2 显示了使用客户端驱动的协调与服务器驱动的方法相比,在第 99.9 个百分位处的延迟改进和在 24 小时内观察到的平均值。如表中所示,客户端驱动的协调方法将 99.9% 延迟的延迟减少了至少 30 毫秒,并将平均延迟减少了 3 到 4 毫秒。延迟的改进是因为客户端驱动的方法消除了负载平衡器的开销以及在将请求分配给随机节点时可能产生的额外网络跃点。如表中所示,平均延迟往往明显低于第 99.9 个百分位的延迟。这是因为 Dynamo 的存储引擎缓存和写入缓冲区具有良好的命中率。此外,由于负载平衡器和网络为响应时间引入了额外的可变性,因此 99.9 的响应时间增益更高

百分位高于平均水平。

6.5 平衡后台与前台任务每个节点除了正常的前台 put/get 操作外,还执行不同类型的后台任务,用于副本同步和数据切换（由于提示或添加/删除节点）。

在早期的生产环境中,这些后台任务会引发资源争用问题并影响常规 put 和 get 操作的性能。因此,有必要确保后台任务仅在常规关键操作未受到显着影响时运行。为此,后台任务与准入控制机制相结合。每个后台任务都使用这个控制器来保留资源（例如数据库)的运行时切片,在所有后台任务中共享。采用基于监控的前台任务性能的反馈机制来更改后台任务可用的切片数量。

准入控制器在执行“前台”放置/获取操作时不断监视资源访问的行为。监控的方面包括磁盘操作的延迟、由于锁争用和事务超时导致的数据库访问失败以及请求队列等待时间。此信息用于检查给定跟踪时间窗口中的延迟（或故障)百分位数是否接近所需阈值。例如,后台控制器检查第 99 个百分位数据库读取延迟（过去 60 秒)与预设阈值（例如 50 毫秒)的接近程度。控制器使用此类比较来评估前台操作的资源可用性。随后,它决定有多少时间片可用于后台任务,从而使用反馈循环来限制后台活动的侵入性。请注意,在 [4] 中研究了管理后台任务的类似问题。

6.6 讨论本节总结了在
Dynamo 的实施和维护过程中获得的一些经验。

在过去的两年中,许多 Amazon 内部服务都使用了 Dynamo,它为其应用程序提供了高水平的可用性。特别是,应用程序已收到 99.9995% 的请求的成功响应（没有超时）,并且迄今为止没有发生数据丢失事件。

此外,Dynamo 的主要优势在于它提供了必要的旋钮,使用 (N,R,W) 的三个参数来根据需要调整实例。与流行的商业数据存储不同,Dynamo 暴露了数据一致性和协调逻辑问题给开发商。一开始,人们可能期望应用程序逻辑变得更加复杂。然而,从历史上看,亚马逊的平台是为高可用性而构建的,许多应用程序旨在处理不同的故障模式和可能出现的不一致。因此,移植此类应用程序以使用 Dynamo 是一项相对简单的任务。对于想要使用 Dynamo 的新应用程序,需要在开发的初始阶段进行一些分析,以选择适合业务案例的正确冲突解决机制。

最后,Dynamo 采用完全会员模型,其中每个节点都知道其对等节点托管的数据。为此,每个节点都主动与系统中的其他节点闲聊完整的路由表。该模型适用于包含数百个节点的系统。但是,将这样的设计扩展为运行数万个节点并非易事,因为维护路由表的开销随着系统大小的增加而增加。这个限制可以通过向 Dynamo 引入分层扩展来克服。另外,请注意,这个问题是由 O(1) DHT 系统积极解决的（例如,[14]）。

7. 结论本文描述了 Dynamo,一
个高度可用和可扩展的数据存储,用于存储 Amazon.com 电子商务平台的一些核心服务的状态。 Dynamo 提供了所需的可用性和性能水平,并成功地处理了服务器故障、数据中心故障和网络分区。 Dynamo 具有增量可扩展性,允许服务所有者根据他们当前的情况进行扩展和缩减

请求负载。 Dynamo 允许服务所有者通过调整参数 N、R 和 W 来定制他们的存储系统，以满足他们所需的性能、持久性和一致性 SLA。

过去一年 Dynamo 的生产使用表明，可以结合分散技术来提供一个单一的高可用性系统。它在最具挑战性的应用程序环境之一中取得的成功表明，最终一致的存储系统可以成为高可用性应用程序的构建块。

致谢

作者要感谢 Pat Helland 对 Dynamo 初始设计的贡献。我们还要感谢 Marvin Theimer 和 Robert van Renesse 的评论。

最后，我们要感谢我们的牧羊人 Jeff Mogul 在准备相机就绪版本时的详细评论和投入，大大提高了论文的质量。

参考文献

[1] Adya, A.,
Bolosky, WJ, Castro, M., Cermak, G., Chaiken, R., Douceur, JR, Howell, J., Lorch, JR, Theimer, M. 和 Wattenhofer, RP 2002. Farsite：为不完全受信任的环境提供联合、可用和可靠的存储。

SIGOPS 操作。系统。修订版36，SI（2002 年 12 月），1-14。

[2] Bernstein, PA 和 Goodman, N.
复制分布式数据库中的并发控制和恢复。 ACM 翻译。关于数据库系统，9(4):596-615，1984 年 12 月

[3] Chang, F.、Dean, J.、Ghemawat, S.、Hsieh, WC、Wallach, DA、Burrows, M.、Chandra, T.、Fikes, A. 和 Gruber, R.
E. 2006. Bigtable：结构化数据的分布式存储系统。在第 7 届 USENIX 操作系统设计和实施研讨会上的会议记录 - 第 7 卷（西雅图，华盛顿州，2006 年 11 月 6 日至 8 日）。 USENIX 协会，加利福尼亚州伯克利，15-15。

[4] Douceur, JR 和 Bolosky, WJ 2000。基于过程
低重要性过程的监管。 SIGOPS 操作。系统。
修订版 34，2（2000 年 4 月），26-27。

[5] Fox，A.，Gribble，SD，Chawathe，Y.，布鲁尔，EA 和
Gauthier, P. 1997。基于集群的可扩展网络服务。
在第十六届 ACM 操作系统原理研讨会论文集上（法国圣马洛，1997 年 10 月 5 日至 8 日）。 WM韦特，埃德。 SOSP 97。 ACM 出版社，纽约，纽约，78-91。

[6] Ghemawat, S.、Gobioff, H. 和 Leung, S. 2003。谷歌文件系统。在第十九届 ACM 操作系统原理研讨会论文集上（美国纽约州博尔顿兰丁，2003 年 10 月 19 日至 22 日）。 SOSP 03. ACM 出版社，纽约，纽约，29-43。

[7] Gray, J.、Helland, P.、O Neil, P. 和 Shasha, D. 1996。复制的危险和解决方案。在1996 年 ACM SIGMOD 国际数据管理会议论文集（蒙特利尔，魁北克，加拿大，1996 年 6 月 4 日至 6 日）。 J. Widom，埃德。西格莫德 96。 ACM 出版社，纽约，纽约，173-182。

[8] Gupta, I.、Chandra, TD 和 Goldszmidt, GS 2001。关于可扩展和高效的分布式故障检测器。在第 20 届 ACM 年度研讨会论文集上
分布式计算原理（美国罗德岛纽波特）。 PODC 01。 ACM 出版社，纽约，纽约，170-179。

[9] Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Wells, C. 和 Zhao, B. 2000。OceanStore：全球规模持久存储架构。 SIGARCH 计算机。

建筑师。新闻28, 5（2000 年 12 月），190-201。

[10] Karger, D.、Lehman, E.、Leighton, T.、Panigrahy, R.、Levine, M. 和 Lewin, D. 1997。一致的散列和随机树：用于缓解网络热点的分布式缓存协议全球资讯网。在第 29 届 ACM 计算理论年度研讨会论文集上（美国德克萨斯州埃尔帕索，1997 年 5 月 4 日至 6 日）。斯托克 97。 ACM 出版社，纽约，纽约，654-663。

[11] 林赛，BG，等。 al.，“分布式数据库注释”，
研究报告 RJ2571(33471)，IBM 研究院，1979 年 7 月

[12] Lamport, L. 分布式系统中的时间、时钟和事件排序。 ACM Communications, 21(7), pp. 558-
565, 1978.

[13] Merkle, R. 基于传统的数字签名
加密功能。 CRYPTO 会议记录，第 369 页–
378. 施普林格出版社，1988 年。

[14] Ramasubramanian, V. 和 Sirer, EG Beehive：对等覆盖中幂律查询分布的 O(1) 查找性能。在第 1 届网络系统设计和实施研讨会上，旧金山，加利福尼亚州，2004 年 3 月 29 日至 31 日。

[15] Reiher, P.、Heidemann, J.、Ratner, D.、Skinner, G. 和 Popek, G. 1994。解决 Ficus 文件系统中的文件冲突。在USENIX 1994 年夏季技术会议论文集上 USENIX 1994 年夏季技术会议 - 第 1 卷（马萨诸塞州波士顿，1994 年 6 月 6 日至 10 日）。 USENIX 协会，伯克利，CA，12-12..

[16] Rowstron, A. 和 Druschel, P. Pastry：可扩展，
用于大规模点对点系统的分散对象定位和路由。中间件会议记录，第 329-350 页，2001 年 11 月。

[17] Rowstron, A. 和 Druschel, P. PAST 中的存储管理和缓存，这是一种大规模、持久的对等存储实用程序。操作系统原理研讨会论文集，2001 年 10 月。

[18] Saito, Y.、Frølund, S.、Veitch, A.、Merchant, A. 和 Spence, S. 2004。FAB：从商品组件构建分布式企业磁盘阵列。 SIGOPS 操作。系统。 Rev. 38, 5（2004 年 12 月），48-58。

[19] Satyanarayanan, M., Kistler, JJ, Siegel, EH Coda：弹性分布式文件系统。 IEEE 工作站操作系统研讨会，1987 年 11 月。

[20] Stoica, I.、Morris, R.、Karger, D.、Kaashoek, MF 和 Balakrishnan, H. 2001。Chord：适用于互联网应用程序的可扩展点对点查找服务。在2001 年计算机通信应用、技术、体系结构和协议会议论文集中

（美国加利福尼亚州圣地亚哥）。 SIGCOMM 01. ACM 出版社，纽约，纽约，149-160。

[21] Terry, DB, Theimer, MM, Petersen, K., Demers, AJ, Spreitzer, MJ 和 Hauser, CH 1995。管理弱连接复制存储系统 Bayou 中的更新冲突。在第十五届 ACM 操作系统原理研讨会论文集上（美国科罗拉多州铜山，1995 年 12 月 3 日至 6 日）。 MB琼斯，埃德。

SOSP 95。 ACM 出版社，纽约，纽约，172-182。

[22] Thomas, RH
多副本数据库的并发控制。数据库系统上的 ACM 事务 4 (2): 180-209, 1979。

[23] Weatherspoon, H.、Eaton, P.、Chun, B. 和 Kubiatowicz, J. 2007. 古代：利用安全日志进行广域分布式存储。 SIGOPS 操作。系统。修订版41, 3（2007 年 6 月），371-384。

[24] Welsh, M.、Culler, D. 和 Brewer, E. 2001。SEDA：一个良好的、可扩展的互联网服务架构。
在第十八届 ACM 操作系统原理研讨会论文集上（加拿大阿尔伯塔省班夫，2001 年 10 月 21 日至 24 日）。
SOSP 01. ACM 出版社，纽约，纽约，230-243。