Dynamo：亚马逊的高可用键值存储

朱塞佩-德坎迪亚、德尼兹-哈斯托伦、马丹-詹帕尼、古纳瓦丹-卡库拉帕蒂、阿维纳什-拉克什曼、亚历克斯-皮尔钦、斯瓦米纳坦-西瓦苏布拉曼尼安、彼得-沃斯霍尔和维尔纳-沃格尔斯

亚马逊网站

摘要

作为全球最大的电子商务运营公司之一，Amazon.com 面临的最大挑战之一就是大规模的可靠性；即使是最轻微的故障也会造成严重的经济后果，并影响客户的信任度。Amazon.com 平台为全球许多网站提供服务，它是在由分布在全球许多数据中心的数以万计的服务器和网络组件组成的基础设施之上实施的。在这种规模下，大大小小的组件都会不断发生故障，而面 对 这些故障时的持久状态管理方式则决定了软件系统的可靠性和可扩展性。

本文介绍了 Dynamo 的设计与实现。Dynamo 是一种高可用性的键值存储系统，亚马逊的一些核心服务使用它来提供 "始终在线 "的体验。为了达到这种可用性水平，Dynamo 牺牲了某些故障情况下的一致性。它广泛使用对象版本控制和应用程序辅助冲突解决，为开发人员提供了新颖的使用界面。

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。

SOSP'07, 2007 年 10 月 14-17 日，美国华盛顿州史蒂文森。

1. 引言

亚马逊运行着一个全球电子商务平台，在高峰期使用分布在全球多个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行要求，而且为了支持持续增长，平台需要具有高度的可扩展性。可靠性是最重要的要求之一，因为即使是最轻微的故障也会造成严重的经济后果，并影响客户的信任。此外，为支持持续增长，平台还需要具有高度可扩展性。

我们的组织从亚马逊平台的运营中学到的经验之一是，系统的可靠性和可扩展性取决于如何管理其应用状态。亚马逊使用的是高度分散、松散耦合、面向服务的架构，由数百个服务组成。在这种环境下，特别需要始终可用的存储技术。例如，即使磁盘出现故障、网络路由出现问题或数据中心被龙卷风摧毁，客户也应该能够查看并向购物车中添加商品。因此，负责管理购物车的服务要求能够随时写入和读取其数据存储，而且其数据需要在多个数据中心之间可用。

在由数百万个组件组成的基础设施中处理故障是我们的标准操作模式；在任何特定时间，总有少量但数量可观的服务器和网络组件发生故障。因此，亚马逊软件系统的构建方式需要将故障处理视为正常情况，而不会影响可用性或性能。

为了满足可靠性和扩展性的需求，亚马逊开发了一系列存储技术，其中最著名的可能是亚马逊简单存储服务（Amazon Simple Storage Service，也可在亚马逊之外使用，称为 Amazon S3）。本文介绍了 Dynamo 的设计与实现，它是为亚马逊平台构建的另一种高可用性、可扩展的分布式数据存储。Dynamo 用于管理对可靠性要求极高的服务状态，这些服务需要严格控制可用性、一致性、成本效益和性能之间的权衡。亚马逊平台上的应用程序种类繁多，对存储的要求也各不相同。这些精选的应用程序需要一种足够灵活的存储技术，让应用程序设计人员能够根据这些权衡适当配置数据存储，从而以最具成本效益的方式实现高可用性和有保证的性能。

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，如提供畅销书列表、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 提供了一个简单的主键接口，可满足这些应用的要求。

Dynamo 综合利用了各种众所周知的技术来实现可扩展性和可用性：数据使用一致散列[10]进行分区和复制，并通过对象版本化[12]促进一致性。在更新过程中，副本之间的一致性是通过类似法定人数的技术和分散式副本同步协议来维持的。Dynamo 采用

是一种基于流言的分布式故障检测和成员协议。Dynamo 是一个完全分散的系统，人工管理需求极低。存储节点可以从 Dynamo 中添加或移除，无需任何手动分区或重新分配。

在过去的一年中，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期间，它能够高效地扩展到极端峰值负载，而不会出现任何停机。例如，维护购物车的服务（购物车服务）在一天内提供了数千万次请求，导致超过 300 万次结账，而管理会话状态的 服务则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在于评估了如何将不同的技术结合起来，以提供单一的高可用性系统。它证明了最终一致性存储系统可用于要求苛刻的生产应用。此外，它还深入探讨了如何调整这些技术，以满足对性能要求非常严格的生产系统的需求。

本文的结构如下。第 2 节介绍背景，第 3 节介绍相关工作。第 4 节介绍系统设计，第 5 节介绍系统实现。第 6 节详细介绍了在生产中运行 Dynamo 所获得的经验和启示，第 7 节对本文进行了总结。在本文中，有许多地方本可以提供更多信息，但为了保护亚马逊的商业利益，我们不得不减少某些细节。因此，第 6 节中的数据中心内部和数据中心之间的延迟、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都是通过综合度量而非绝对细节提供的。

2. 背 景

亚马逊的电子商务平台由数百项服务组成，这些服务协同工作，提供从推荐、订单执行到欺诈检测等各种功能。每项服务都通过定义明确的接口公开，并可通过网络访问。这些服务由分布在全球多个数据中心的数万台服务器组成的基础设施托管。这些服务中有些是无状态服务（即汇总其他服务响应的服务），有些是有状态服务（即通过在持久存储中存储的状态执行业务逻辑来生成响应的服务）。

传统上，生产系统将其状态存储在关系数据库中。然而，对于许多更常见的状态持久性使用模式来说，关系数据库并不是一个理想的解决方案。这些服务大多只通过主键存储和检索数据，不需要关系数据库管理系统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，因此是一种非常低效的解决方案。此外，现有的复制技术也很有限，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，但要扩展数据库或使用智能分区方案来实现负载平衡仍非易事。

本文介绍的 Dynamo 是一种高度可用的数据存储技术，可满足这些重要服务类型的需求。Dynamo 有一个简单的键/值接口，具有明确定义的一致性窗口，可用性高，资源使用效率高，并有一个简单的扩展方案来应对数据集大小或请求率的增长。使用 Dynamo 的每个服务都运行自己的 Dynamo 实例。

2.1 系统假设和要求

这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）。没有跨越多个数据项的操作，也不需要关系模式。亚马逊的大部分服务都可以使用这种简单的查询模型，而且不需要任何关系模式。Dynamo 面向需要存储相对较小（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*）是一组保证数据库事务得到可靠处理的属性。在数据库中，对数据进行的单个逻辑操作称为事务。亚马逊的经验表明，提供 ACID 保证的数据存储往往可用性较差。这一点已得到业界和学术界的广泛认可[5]。Dynamo 针对的是以较弱一致性（ACID 中的 "C"）运行的应用程序，如果这样能带来高可用性的话。Dynamo 不提供任何隔离保证，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。在亚马逊平台上，服务有严格的延迟要求，一般以分布的 99.9^{th} 百分位数来衡量。鉴于状态访问在服务运行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 要求（参见下文第 2.2 节）。服务必须能够对 Dynamo 进行配置，从而持续满足其延迟和吞吐量要求。这需要在性能、成本效率、可用性和耐用性保证方面进行权衡。

其他假设：Dynamo 仅供亚马逊内部服务使用。其运行环境被假定为非敌对环境，并且不存在与安全相关的要求，如身份验证和授权。此外，由于每个服务都使用各自独立的 Dynamo 实例，因此其初始设计的目标规模可达数百台存储主机。我们将在后面的章节中讨论 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)

为了保证应用程序能在限定的时间内交付其功能，平台中的每个依赖项都需 要 在更严格的限定时间内交付其功能。客户和服务需要签订服务级别协议（SLA），这是一份经过正式协商的合同，客户和服务在合同中就多个系统相关特性达成一致，其中最主要的包括客户对特定应用程序接口的预期请求率分布以及在这些条件下的预期服务延迟。简单 SLA 的一个例子是服务保证它将

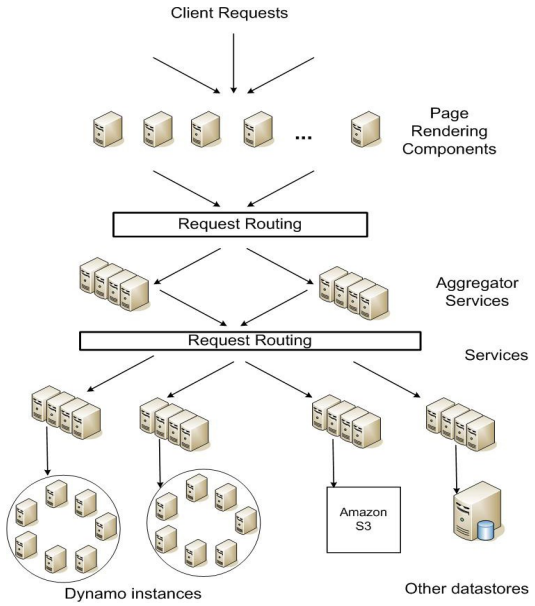


图 1：亚马逊平台面向服务的架构

在每秒 500 个请求的峰值客户端负载下，在 300 毫秒内对 99.9% 的请求做出响应。

在亚马逊面向服务的分散式基础设施中，SLA 发挥着重要作用。例如，向某个电子商务网站提出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响应。这些服务通常有多个依赖关系，而这些依赖关系往往是其他服务，因此应用程序的调用图往往不止一个层次。为确保页面渲染引擎能对页面交付保持明确的约束，调用链中的每个服务都必须遵守其性能合约。

图 1 显示了亚马逊平台架构的抽象视图，其中动态网络内容由页面渲染组件生成，而页面渲染组件又会查询许多其他服务。一个服务可以使用不同的数据存储来管理其状态，这些数据存储只能在其服务边界内访问。有些服务充当聚合器，使用多个其他服务生成复合响应。通常情况下，聚合器服务是无状态的，尽管它们使用大量缓存。

业内形成以性能为导向的 SLA 的常见方法是使用平均值、中位数和预期方差进行描述。在亚马逊，我们发现如果我们的目标是建立一个**所有**客户都能获得良好体验的系统，而不仅仅是大多数客户，那么这些指标就不够好。例 如 ， 如果使用广泛的个性化技术，那么历史记录较长的客户就需要更多的处理，这就会影响分布高端的性能。以平均响应时间或中位响应时间表示的 SLA 无法解决这一重要客户群的性能问题。为了解决这个问题，在亚马逊，SLA 在以下方面进行表达和

衡量

99.9th 分布百分位数。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，要提高这么高的 性能，成本会大幅增加。亚马逊的经验

生产系统表明，与那些满足基于平均值或中位数定义的SLA 的系统相比，这种方法能提供更好的整体体验。

本文多次提到 99.9^{th} 百分位数分布，这反映了亚马逊工程师从客户体验的角度出发，对性能的不懈关注。许多论文报告的都是平均值，因此在有必要进行比较时，我们也会将其包括在内。然而，亚马逊的工程和优化工作并不专注于平均值。有几项技术，如负载平衡选择写协调器，完全是为了将性能控制在 99.9^{th} 百分位数。

存储系统通常在建立服务的 SLA 方面扮演重要角色，尤其是在业务逻辑相对轻量级的情况下，亚马逊的许多服务就是如此。因此，状态管理就成了服务 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，如耐用性和一致性，并让服务在功能、性能和成本效益之间自行权衡。

2.3 设计考虑因素

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界面。为了实现这种一致性，这些算法不得不在某些故障情况下权衡数据的可用性。例如，与其处理答案正确性的不确定性，不如在绝对确定答案正确之前让数据不可用。众所周知，从早期的复制数据库工作来看，在处理网络故障的可能性时，无法同时实现强一致性和高数据可用性[2, 11]。因此，系统和应用程序需要了解在哪些条件下可 以 实现哪些特性。

对于易受服务器和网络故障影响的系统，可以通过使用乐观复制技术来提高可用性，即允许更改在后台传播到副本，并容忍并发的、断开的工作。这种方法面 临 的 挑战是，它可能会导致必须检测和解决的冲突变更。冲突解决过程会带来两个问题：何时解决和由谁来解决。Dynamo 被设计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因素是决定*何时*执行解决更新冲突的过程，即冲突是在读取过程中解决还是在写入过程中解决。许多传统数据存储在写入过程中执行冲突解决，并保持读取复杂性的简单性[7]。在这类系统中，如果数据存储无法在给定时间内到达所有（或大部分）副本，写入可能会被拒绝。另一方面，Dynamo 的 目标设计空间是 "始终可写 "数据存储（即对写操作高度可用的数据存储）。对于许多亚马逊服务来说，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网络和服务器故障的情况下添加和删除购物车中的物品。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*由谁来*执行冲突解决过程。这可以由数据存储或应用程序完成。如果冲突解决由数据存储完成，其选择就相当有限。在这种情况下，数据存储只能使用简单的策略（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 "冲突版本，并返回一个统一的购物车。尽管有这样的灵活性，但有些应用程序开发人员可能并不想编写自己的冲突解决机制，而是选择将其推送给数据存储，而数据存储则会选择 "最后写入者获胜 "这样的简单策略。

设计中采用的其他主要原则包括

递增可扩展性：Dynamo 应该能够一次扩展一个存储主机（以下简称为 "*节点*"），并将对系统操作员和系统本身的影响降到最低。

对称性：Dynamo 中的每个节点都应与其对等节点承担相同的责任；不应存在特殊节点或承担特殊角色或额外责任的节点。根据我们的经验，对称性简化了系统调配和维护的过程。

去中心化：作为对称性的延伸，设计应倾向于分散的点对点技术，而不是集中控制。在过去，集中式控制会导致系统中断，而我们的目标是尽可能避免这种情况。这将带来一个更简单、更可扩展、更可用的系统。

异构性：系统需要能够利用其所运行的基础设施的异构性，例如，工作分配必须与单个服务器的能力成正比。这对于在无需同时升级所有主机的情况下添加容量更大的新节点至关重要。

3. 相关工作

3.1 点对点系统

有几种点对点（P2P）系统研究了数据存储和分发问题。第一代 P2P 系统，如 Freenet 和 Gnutella[1]，主要用作文件共享系统。这些都是非结构化 P2P 网络的例子，在这些网络中，对等点之间的重叠链接是任意建立的。在这些网络中，搜索查询通常会在网络中泛滥，以找到尽可能多的共享数据的对等点。P2P 系统发展到下一代，成为广为人知的结构化 P2P 网络。这些网络采用全局一致的协议，确保任何节点都能有效地将搜索查询路由到拥有所需数据的对等节点。Pastry [16] 和 Chord [20] 等系统使用路由机制来确保在一定的跳数范围内回答查询。为了减少多跳路由带来的额外延迟，一些 P2P 系统（如 [14]）采用了 O(1) 路由机制，即每个对等点在本地维护足够的路由信息，这样它就能在跳数不变的情况下将请求（访问数据项）路由到合适的对等点。

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

Oceanstore [9] 和 PAST [17] 等各种存储系统都是在这些路由覆盖的基础上构建的。Oceanstore 提供全局、事务性、持久存储服务，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，它使用了一种基于冲突解决的更新模型。冲突解决是在 [21] 中引入的，目的是减少事务中止的次数。Oceanstore 解决冲突的方法是处理一系列更新，在其中选择一个总顺序，然后按原子顺序应用这些更新。Oceanstore 适用于在不信任的基础设施上复制数据的环境。相比之下，PAST 在 Pastry 的基础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上建立必要的存储语义（如可变文件）。

3.2 分布式文件系统和数据库

文件系统和数据库系统界对分布式数据的性能、可用性和耐用性进行了广泛研究。与只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。Ficus [15] 和 Coda [19] 等系统以牺牲一致性为代价，复制文件以获得高可用性。更新冲突通常使用专门的冲突解决程序进行管理。Farsite 系统[1]是一种分布式文件系统，它不使用任何像 NFS 这样的集中式服务器。Farsite 通过复制实现了高可用性和可扩展性。谷歌文件系统[6]是另一种分布式文件系统，用于托管谷歌内部应用程序的状态。GFS 采用简单的设计，由一个主服务器托管整个元数据，数据被分割成块并存储在块服务器中。Bayou 是一种分布式关系数据库系统，允许断开操作并提供最终数据一致性[21]。

在这些系统中，Bayou、Coda 和 Ficus 允许断开操作，并能应对网络分区和中断等问题。这些系统的冲突解决程序各不相同。例如，Coda 和 Ficus 执行系统级冲突解决，而 Bayou 允许应用级冲突解决。不过，它们都能保证最终的一致性。与这些系统类似，Dynamo 允许在网络分区期间继续进行读写操作，并使用不同的冲突解决机制解决更新后的冲突。分布式块存储系统（如 FAB [18]）将大型对象分割成较小的块，并以高度可用的方式存储每个块。与这些系统相比，键值存储更适合本例，因为：(a) 键值存储旨在存储相对较小的对象（大小小于 1M）；(b) 键值存储更容易按应用进行配置。Antiquity 是一种广域分布式存储系统，设计用于处理多个服务器故障[23]。它使用安全日志来保持数据的完整性，在多个服务器上复制每个日志以保证数据的持久性，并使用拜占庭容错协议来确保数据的一致性。与 Antiquity 不同，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。Bigtable 是一种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地图，允许应用程序使用多种属性访问数据[2]。与 Bigtable 相比，Dynamo 针对的是只需要键/值访问的应用程序，主要侧重于高可用性，即使在网络分区或服务器故障的情况下也不会拒绝更新。

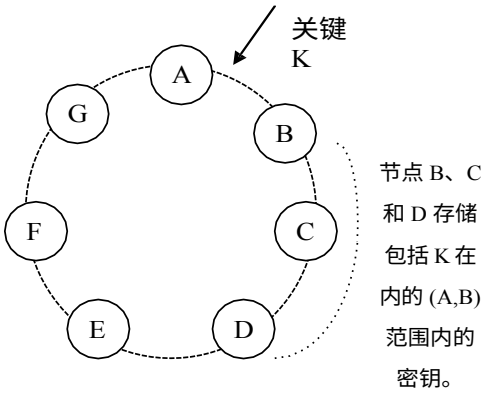


图 2：Dynamo Ring 中密钥的分区和复制。

传统的复制关系型数据库系统主要解决保证复制数据强一致性的问题。虽然强一致性为应用程序编写者提供了一个方便的编程模型，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一致性保证。

3.3 讨论

Dynamo 与上述分散式存储系统的目标要求不同。首先，Dynamo 主要面向需要 "始终可写 "数据存储的应用程序，在这种存储中，更新不会因故障或并发写入而被拒绝。这是许多亚马逊应用程序的关键要求。其次，如前所述，Dynamo 是为单一管理域内的基础架构而构建的，该管理域中的所有节点都被假定为可信的。第三，使用 Dynamo 的应用程序不需要分层命名空间（许多文件系统的规范）或复杂关系模式（传统数据库支持）的支持。第四，Dynamo 专为对延迟敏感的应用而构建，这些应用要求至少 99.9% 的读写操作在几百毫秒内完成。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 等多个分布式哈希表系统采用的典型设计）。这是因为多跳路由会增加响应时间的不稳定性，从而增加较高百分位数的延迟。Dynamo 的特点是零跳 DHT，每个节点都在本地维护足够的路由信息，以便将请求直接路由到相应的节点。

4. 系统架构

需要在生产环境中运行的存储系统的架构非常复杂。除了实际的数据持久化组件外，系统还需要为负载平衡、成员资格和故障检测、故障恢复、副本同步、过载处理、状态传输、并发和作业调度、请求拦截、请求路由、系统监控和报警以及配置管理提供可扩展且稳健的解决方案。我们不可能对每种解决方案都进行详细描述，因此本文将重点介绍 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* 中使用的技术及其优势汇总。

问题	技术	优势
分区	一致的散列	增量可扩展性
写入的高可用性	读取时调节矢量时钟	版本大小与更新率脱钩。
处理临时故障	马虎的法定人数和暗示的移交	当部分副本不可用时，可提供高可用性和耐用性保证。
从永久性故障中恢复	利用梅克尔树反熵	在后台同步不同的副本。
成员资格和故障检测	基于流言的成员协议和故障检测	保持对称性，避免使用中央注册表来存储成员信息和节点有效性信息。

表 1 概述了 Dynamo 使用的技术及其各自的优势。

4.1 系统界面

Dynamo 通过一个简单的接口存储与 key 关联的对象；它提供两种操作：get() 和 put()。get(*key*)操作会定位存储系统中与*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*对象副本的*位置，并将副本写入磁盘。*上下文*对调用者不透明的对象系统元数据进行了编码，其中包括对象的版本等信息。上下文信息与对象一起存储，以便系统能验证在放置请求中提供的上下文对象的有效性。

Dynamo 将调用者提供的密钥和对象都视为不透明的字节数组。它会对密钥进行 MD5 哈希运算，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必须逐步扩展。这就需要一种机制，将数据动态地划分到系统中的节点（即存储主机）上。Dynamo 的分区方案依靠一致散列在多个存储主机上分配负载。在一致散列[10]中，散列函数的输出范围被视为一个固定的圆形空间或 "环"（即最大散列值环绕到最小散列值）。系统中的每个节点都会在这个空间内被分配一个随机值，代表其在环上的 "位置"。每个由密钥标识的数据项都会分配给一个节点，方法是对数据项的密钥进行散列，得出其在环上的位置，然后顺时针走环，找到第一个位置大于数据项位置的节点。

因此，每个节点都要对它和它在环上的前一个节点之间的环上区域负责。一致散列的主要优点是，节点的离开或到达只会影响其近邻，其他节点不受影响。

基本的一致散列算法面临着一些挑战。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 使用了一致散列的一种变体（类似于 [10, 20] 中使用的变体）：不是将节点映射到圆环中的一个点，而是将每个节点分配到圆环中的多个点。为此，Dynamo 使用了 "虚拟节点 "的概念。虚拟节点看起来就像系统中的一个节点，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，它会被分配到环中的多个位置（因此称为 "代币"）。微调 Dynamo 分区方案的过程将在第 6 节中讨论。

使用虚拟节点有以下优点：

- 如果某个节点不可用（由于故障或日常维护），则该节点处理的负载会平均分配给其余可用节点。
- 当一个节点再次可用或系统中添加了一个新节点时，新的可用节点会从其他每个可用节点接受大致相等的负载量。
- 一个节点负责的虚拟节点数量可根据其容量决定，并考虑物理基础设施的异构性。

4.3 复制

为了实现高可用性和耐用性，Dynamo 在多个主机上复制数据。每个数据项在 N 台主机上复制，其中 N 是 *"按实例 "*配置的参数。每个密钥 *k* 都分配给一个协调器节点（如上一节所述）。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，协调器还将这些密钥复制到环中 N-1 个顺时针方向的后续节点上。这样就形成了一个系统，每个节点负责它与其 N 个th 前置节点之间的环状区域。在图 2 中，节点 B 除了在本地存储密钥 *k 外，还*将其复制到节点 C 和 D。节点 D 将存储范围为 (A，B]、(B，C] 和 (C，D] 的密钥。

负责存储特定密钥的节点列表称为*首选列表*。正如第 4.8 节所解释的，系统的设计使系统中的每个节点都能为任何特定密钥确定哪些节点应在此列表中。为了考虑到节点故障，首选列表包含 N 个以上的节点。需要注意的是，由于使用了虚拟节点，特定密钥的前 N 个后继位置有可能被少于 N 个不同的物理节点所拥有（即一个节点可能拥有前 N 个位置中的不止一个）。为了解决这个问题，在构建密钥的优先级列表时，会跳过环中的位置，以确保列表只包含不同的物理节点。

4.4 数据版本管理

Dynamo 提供最终一致性，允许将更新异步传播到所有副本。调用 put() 可以

在更新应用到所有副本之前，get() 返回给调用者，这可能会导致后续 get() 操作返回的对象没有最新更新。如果不发生故障，更新传播时间就会受到限制。但是，在某些故障情况下（如服务器中断或网络分区），更新可能会在很长一段时间内无法到达所有副本。

亚马逊平台上有一类应用程序可以容忍这种不一致性，并且可以在这些条件下运行。例如，购物车应用程序要求 "*添加到购物车* "操作永远不能被遗忘或拒绝。如果购物车的最新状态不可用，而用户对购物车的旧版本进行了更改，则该更改仍有意义，应予以保留。但与此同时，它也不应该取代购物车当前不可用的状态，因为购物车本身也可能包含应该保留的更改。请注意，"*添加物品到购物车* "和 "*从购物车中删除物*品 "这两个操作都是向 Dynamo 提出的 put 请求。当客户要将物品添加到购物车（或从购物车中删除），而最新版本不可用时，物品会被添加到旧版本（或从旧版本中 删除），不同的版本会在稍后进行调和。

为了提供这种保证，Dynamo 将每次修改的结果都视为数据的一个新的不可变版本。它允许一个对象的多个版本同时存在于系统中。大多数情况下，新版本会取代之前的版本，系统本身也能确定权威版本（语法调和）。但是，在出现故障和并发更新的情况下，可能会出现版本分支，导致对象的版本相互冲突。在这种情况下，系统无法调和同一对象的多个版本，客户端必须执行调和操作，以便将数据演化的多个分支*折叠*为一个分支（语义调和）。折叠操作的一个典型例子是 "合并 "客户购物车的不同版本。使用这种调和机制，"添加到购物车 "操作永远不会丢失。但是，被删除的项目可能会重新出现。

重要的是要明白，某些故障模式可能会导致系统中同一数据不仅有两个版本，而且有多个版本。在网络分区和节点故障的情况下进 行 更新，可能会导致一个对象具有不同的子历史版本，系统需要在未来对其进行调节。这就要求我们在设计应用程序时，明确承认同一数据可能存在多个版本（以避免丢失任何更新）。

Dynamo 使用向量时钟 [12] 来捕捉同一对象不同版本之间的因果关系。向量时钟实际上是一个（节点、计数器）对列表。每个对象的每个版本都有一个向量时钟。我们可以通过检查一个对象的两个版本的矢量时钟，来确定它们是否处于平行分支上或是否有因果关系。如果第一个对象时钟上的计数器小于或等于第二个时钟上的所有节点，那么第一个对象就是第二个对象的祖先，可以被遗忘。否则，这两个变化会被认为是冲突的，需要调和。

在 Dynamo 中，当客户端希望更新对象时，必须指定要更新的版本。具体做法是传递从先前读取操作中获得的上下文，其中包含矢量时钟信息。处理读取请求时，如果

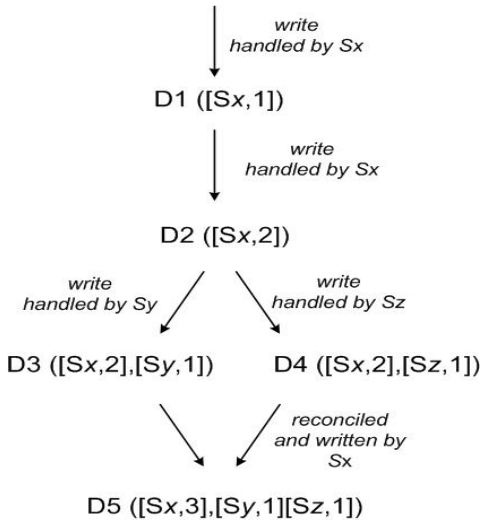


图 3：对象的版本随时间演变。

如果 Dynamo 可以访问多个在语法上无法调和的分支，它将返回叶子上的所有对象，并在上下文中提供相应的版本信息。使用此上下文进行的更新被视为已调和了不同的版本，各分支将合并为一个新版本。

为了说明矢量时钟的使用，让我们来看看图 3 所示的示例。客户端写入一个新对象。处理该键写入的节点（如 Sx）会增加其序列号，并用它来创建数据的矢量时钟。系统现在拥有对象 D1 及其相关时钟 [(Sx, 1)]。客户端更新对象。假设同一节点也处理该请求。系统现在还有对象 D2 及其相关时钟 [(Sx, 2)]。D2 *是* D1 的*后代*，因此会覆盖 D1，但在尚未看到 D2 的节点上可能还有 D1 的副本。假设同一客户端再次更新对象，并由不同的服务器（如 Sy）处理请求。系统现在有数据 D3 及其相关时钟 [(Sx, 2), (Sy, 1)]。

接下来，假设不同的客户端读取了 D2，然后试图更新它，另一个节点（比如 Sz）完成了写入操作。系统现在有了 D4（D2 的后代），其版本时钟为 [(Sx, 2), (Sz, 1)]。知道 D1 或 D2 的节点在接收到 D4 及其时钟后，可以判断 D1 和 D2 已被新数据覆盖，可以进行垃圾回收。知道 D3 并接收到 D4 的节点会发现它们之间没有因果关系。换句话说，D3 和 D4 中的变化并不反映在彼此中。两个版本的数据都必须保留，并（在读取时）呈现给客户端，以便进行语义调节。

现在假设某个客户端同时读取了 D3 和 D4（上下文将反映出读取时发现了这两个值）。读取的上下文是 D3 和 D4 的时钟摘要，即 [(Sx, 2), (Sy, 1), (Sz, 1)]。如果客户端执行对账，节点 Sx 协调写入，Sx 将更新其在 时钟中的序列号。新数据 D5 的时钟如下：[(Sx,3),(Sy,1),(Sz,1)]。

矢量时钟可能存在的一个问题是，如果许多服务器协调写入一个矢量时钟，那么矢量时钟的大小可能会增大。

对象。实际上，这种情况不太可能发生，因为写入请求通常会由优先级列表中排名前 N 的节点之一处理。在网络分区或多个服务器故障的情况下，写入请求可能会由偏好列表中前 N 个节点以外的节点处理，从而导致矢量时钟的大小增加。在这种情况下，最好限制矢量时钟的大小。为此，Dynamo 采用了以下时钟截断方案：在每个（节点、计数器）对中，Dynamo 都会存储一个时间戳，表明节点最后一次更新数据项的时间。当矢量时钟中（节点、计数器）对的数量达到一个阈值（比如 10）时，时钟中最老的一对就会被移除。显然，这种截断方案会导致调节效率低下，因为无法准确推导出后代关系。不过，这个问题还没有在生产中出现过，因此还没有对这个问题进行深入研究。

4.5 执行 get () 和 put () 操作

Dynamo 中的任何存储节点都有资格接收客户端对任何密钥的获取和放入操作。在本节中，为简单起见，我们将介绍在无故障环境下如何执行这些操作，并在随后的章节中介绍在发生故障时如何执行读取和写入操作。

获取和投放操作都是通过 HTTP 使用亚马逊特定于基础设施的请求处理框架调用的。客户端可以使用两种策略来选择节点：(1) 通过通用负载平衡器路由请求，该负载平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，该库可将请求直接路由到适当的协调器节点。第一种方法的优势在于，客户端无需在其应用程序中链接任何 Dynamo 专用代码，而第二种策略由于跳过了潜在的转发步骤，因此可以实现更低的延迟。

处理读取或写入操作的节点称为*协调器*。通常，协调器是优先级列表中前 N 个节点中的第一个。如果请求是通过负载均衡器接收的，那么访问密钥的请求可能会被路由到环中的任意节点。在这种情况下，如果接收请 求 的 节 点 不在所请求的密钥优先级列表的前 N 个节点中，则该节点不会对请求进行协调。相反，该节点会将请求转发给优先级列表前 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，而跳过那些故障或无法访问的节点。当所有节点都健康时，将访问密钥首选项列表中的前 N 个节点。当出现节点故障或网络分区时，就会访问首选项列表中排名靠后的节点。

为了保持副本之间的一致性，Dynamo 使用了与法定人数系统中使用的协议类似的一致性协议。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 是成功写入操作必须参与的最少节点数。将 R 和 W 设为 R + W > N，就会产生一个类似法定人数的系统。在这个模型中，获取（或写入）操作的延迟由 R（或 W）副本中最慢的副本决定。因此，R 和 W 通常配置为小于 N，以提供更好的延迟。

在接收到密钥的 put() 请求后，协调器会生成新版本的矢量时钟，并在本地写入新版本。然后，协调器将新版本（连同

新矢量时钟）给 N 个排名最高的可到达节点。如果至少有 W-1 个节点响应，则认为写入成功。

同样，对于 get()请求，协调器会向该键的优先级列表中排名最高的 N 个可到达节点请求该键的所有现有数据版本，然后等待 R 个响应，再将结果返回给客户端。如果协调器最终收集到多个版本的数据，它会返回所有它认为没有因果关系的版本。然后对不同版本进行调和，并写回取代当前版本的调和版本。

4.6 处理故障：提示切换

如果 Dynamo 使用传统的法定人数方法，那么在服务器故障和网络分区时，它将无法使用，甚至在最简单的故障条件下也会降低耐用性。为了解决这个问题，Dynamo 并不强制执行严格的法定人数成员资格，而是使用 "马虎的法定人数"；所有读写操作都在首选列表中的前 N 个*健康*节点上执行，而这 N 个节点不一定总是在走一致散列环时遇到的前 N 个节点。

请看图 2 中 N=3 的 Dynamo 配置示例。在此示例中，如果在写操作过程中节点 A 暂时宕机或无法访问，那么通常位于 A 节点上的副本将被发送到节点 D。发送到 D 节点的副本将在其元数据中带有一个提示，表明哪个节点是副本的预期接收方（在本例中为 A 节点）。收到提示副本的节点会将其保存在一个单独的本地数据库中，并定期扫描。检测到 A 已恢复后，D 会尝试将副本传送给 A。一旦传送成功，D 可以从本地存储中删除对象，而不会减少系统中的副本总数。

Dynamo 通过使用提示切换功能，确保读写操作不会因临时节点或网络故障而失败。需要最高可用性的应用程序可以将 W 设置为 1，这样就能确保只要系统中的单个节点已将密钥持久写入本地存储，写入请求就会被接受。因此，只有当系统中所有节点都不可用时，写入请求才会被拒绝。不过，在实际生产中，大多数亚马逊服务都会设置更高的 W，以满足所需的耐用性水平。关于配置 N、R 和 W 的更详细讨论将在第 6 节中展开。

高可用性存储系统必须能够处理整个数据中心的故障。断电、冷却故障、网络故障和自然灾害都会导致数据中心发生故障。Dynamo 的配置方式是在多个数据中心之间复制每个对象。从本质上讲，密钥的首选项列表是这样构建的：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，而不会造成数据中断。

4.7 处理永久性故障复制同步

如果系统成员流失率较低，节点故障是短暂的，那么提示移交的效果最好。在某些情况下，提示副本在返回到系统之前就已经不可用了。

原始副本节点。为了应对这种情况和其他对耐久性的威胁，Dynamo 实施了反熵协议（副本同步），以保持副本同步。

为了更快地检测副本之间的不一致，并最大限度地减少传输的数据量，Dynamo 使用了 Merkle 树 [13]。Merkle 树是一棵哈希树，树叶是各个键值的哈希值。树中较高的父节点是其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 树还有助于减少在检查副本间不一致时需要传输的数据量。例如，如果两棵树根部的哈希值相等，那么树中叶子节点的值也相等，节点无需同步。如果不相等，则意味着某些副本的值不同。在这种情况下，节点可以交换子节点的哈希值，这个过程一直持续到树的叶子，这时主机就能识别出 "不同步 "的键。梅克尔树最大限度地减少了同步所需的数据传输量，并减少了反熵过程中的磁盘读取次数。

Dynamo 使用 Merkle 树进行反熵，具体如下：每个节点为其托管的每个密钥范围（虚拟节点覆盖的密钥集）维护一棵单独的 Merkle 树。这样，节点就能比较密钥范围内的密钥是否是最新的。在这种方案中，两个节点交换它们共同托管的密钥范围对应的 Merkle 树的根。随后，节点使用上述树遍历方案确定它们是否存在任何差异，并执行相应的同步操作。这种方案的缺点是，当节点加入或离开系统时，许多密钥范围会发生变化，因此需要重新计算树。不过，第 6.2 节中描述的细化分区方案可以解决这个问题。

4.8 成员资格和故障检测

4.8.1 环形会员制

在亚马逊环境中，节点中断（由于故障和维护任务）通常是短暂的，但可能会持续较长时间。节点中断很少意味着永久性离开，因此不应导致重新平衡分区分配或修复无法到达的副本。同样，手动错误也可能导致无意中启动新的 Dynamo 节点。基于这些原因，我们认为使用一种明确的机制来启动 Dynamo 环上节点的添加和移除是合适的。管理员使用命令行工具或浏览器连接到 Dynamo 节点，并发出成员变更请求，将节点加入环或从环中移除节点。发出请求的节点会将成员资格变更及其发出时间写入持久存储中。成员资格变更会形成历史记录，因为节点可以多次被移除和重新加入。基于流言的协议会传播成员资格变更，并保持成员资格的最终一致性。每个节点每秒与一个随机选择的对等节点联系，两个节点会有效地调和它们持久的成员资格变更历史。

当节点首次启动时，它会选择其标记集（一致散列空间中的虚拟节点），并将节点映射到各自的标记集。映射被持久保存在磁盘和

最初只包含本地节点和标记集。存储在不同 Dynamo 节点的映射会在调和成员变更历史的相同通信交换中进行调和。因此，分区和位置信息也会通过基于流言的协议传播，每个存储节点都知道其对等节点处理的令牌范围。这样，每个节点就能将密钥的读/写操作直接转发给正确的节点集。

4.8.2 外部发现

上述机制可能会暂时导致一个逻辑分区的 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 都会认为自己是环的成员，但都不会立即意识到对方。为了防止逻辑分区，一些 Dynamo 节点扮演了种子的角色。种子节点是通过外部机制发现的节点，所有节点都知道这些节点。由于所有节点最终都会将自己的成员身份与种子核对，因此逻辑分区的可能性极低。种子可以从静态配置或配置服务中获得。种子通常是 Dynamo 环中功能完备的节点。

4.8.3 故障检测

Dynamo 中的失败检测用于避免在 get() 和 put() 操作期间以及在传输分区和提示副本时尝试与不可达的对等节点通信。为了避免失败的通信尝试，纯粹的本地故障检测概念已经完全足够：如果节点 B 不响应节点 A 的消息（即使 B 响应了节点 C 的消息），节点 A 可以认为节点 B 失败了。在 Dynamo 环中，客户端请求会以稳定的速度产生节点间通信，当 B 节点不响应消息时，节点 A 很快就会发现 B 节点没有响应；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，检查后者是否恢复。在没有客户端请求驱动两个节点之间流量的情况下，两个节点都不需要知道对方是否可到达和响应。

分散式故障检测协议使用简单的流言式协议，使系统中的每个节点都能了解其他节点的到达（或离开）情况。关于分散式故障检测器和影响其准确性的参数的详细信息，感兴趣的读者可参阅 [8]。Dynamo 的早期设计使用分散式故障检测器来保持故障状态的全局一致性。后来人们发现，显式节点加入和退出方法不需要故障状态的全局视图。这是因为显式节点加入和退出方法会通知节点永久性的节点添加和删除，而单个节点在无法与其他节点通信时（转发请求时）会检测到临时性的节点故障。

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，它会被分配到一定数量的令牌，这些令牌随机散布在环上。对于分配给节点 X 的每个密钥范围，当前可能有若干节点（小于或等于 N）处于以下状态

当 X 被添加到系统中时，它将负责存储范围为 (F，G]、(G，A] 和 (A，X] 的密钥。因此，节点 B 、C 和 D 不再需要存储这些范围内的密钥。因此，节点 B 、C 和 D 将向 X 提供并在 X 确认后转移相应的密钥集。当一个节点从系统中移除时，密钥的重新分配会以相反的方式进行。

运行经验表明，这种方法可以将密钥分发的负载均匀地分配到各个存储节点上，这对于满足延迟要求和确保快速启动非常重要。最后，通过在源节点和目标节点之间增加一轮确认，可以确保目标节点不会收到给定密钥范围内的任何重复传输。

5. 实施

在 Dynamo 中，每个存储节点都有三个主要软件组件：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。

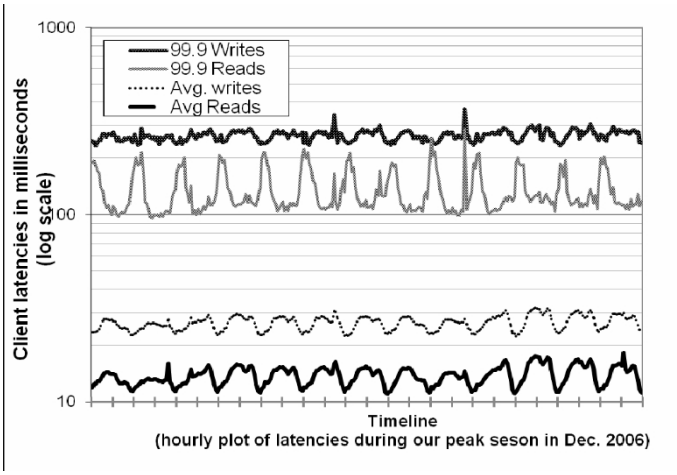
Dynamo 的本地持久化组件允许插入不同的存储引擎。目前使用的引擎有 Berkeley Database (BDB) Transactional Data Store[2] 、BDB Java Edition 、MySQL 和带有持久性后备存储的内存缓冲区。设计可插拔持久化组件的主要原因是选择最适合应用程序访问模式的存储引擎。例如，BDB 可以处负责处理其令牌范围内的密钥。由于
将密钥范围分配给 X 时，一些现有节点不再需要某些密钥，这些节点会将这些密钥转移给 X。

理通常为几十KB 的对象，而 MySQL 可以处理更大的对象。应用程序根据其对象大小分布选择 Dynamo 的本地持久化引擎。Dynamo 的大多数生产实例都使用 BDB 事务数据存储。

请求协调组件建立在事件驱动的消息传递基底之上，消息处理管道被分成多个阶段，与 SEDA 架构[24] 类似。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（读取）或在一个或多个节点存储数据（写入），代表客户端执行读取和写入请求。每个客户端请求都会在接收客户端请求的节点上创建一个状态机。状态机包含识别负责某个密钥的节点、发送请求、等待响应、可能的重试、处理回复并将响应打包给客户端的所有逻辑。每个状态机实例处理一个客户端请求。例如，读取操作会执行以下状态机：(i) 向节点发送读取请求；(ii) 等待所需的最少回应；(iii) 如果在给定时间内收到的回应太少，则请求失败；(iv) 否则，收集所有数据版本并确定要返回的版本；(v) 如果启用了版本控制，则执行语法调节并生成不透明的写入上下文，其中包含包含所有剩余版本的向量时钟。为简洁起见，不包括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



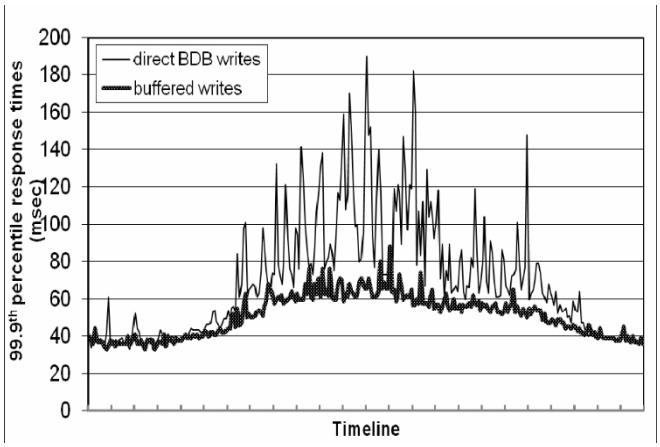


图 4：2006 年 12 月请求高峰期读写请求延迟的平均值和 99.9 百分位数。x 轴上连续刻度线之间的间隔为 12 小时。延迟的昼夜变化规律与请求率相似，99.9 百分位数延迟比平均值高出一个数量级。

机器会等待一小段时间来接收任何未回复。如果有 任何响应返回了过期版本，协调器就会用最新版本更新这些节点。这个过程被称为 *读修复，* 因为它能在适当的时候修复错过最新更新的副本，并使反熵协议不必再做这项工作。

如前所述，写入请求由优先级列表中前 N 个节点中的一个来协调。虽然最好始终由前 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到一个位置，但这种方法会导致负载分布不均，从而违反 SLA。这是因为请求负载在对象间的分布并不均匀。为了解决这个问题，允许优先级列表中排名前 N 个节点中的任何一个节点来协调写入。特别是，由于每次写操作通常都是在读取操作之后进行的，因此写操作的协调者会选择对上一次读操作回复最快的节点，而上一次读操作已存储在请求的上下文信息中。通过这种优化，我们可以选择拥有前一次读取操作所读取数据的节点，从而增加获得 "读写 "一致性的机会。它还降低了请求处理性能的可变性，从而提高了 99.9 百分位数的性能。

6. 经验和教训

Dynamo 由多个不同配置的服务使用。这些实例的版本调节逻辑和读/写法定人数特性各不相同。以下是使用 Dynamo 的主要模式：

- *业务逻辑特定调节：* 这是 Dynamo 的常用用例。每个数据对象都在多个节点上复制。如果版本不同，客户端应用程序将执行自己的调节逻辑。前面讨论的购物车服务就是一个典型的例子。它的业务逻辑通过合并客户购物车的不同版本来调节对象。

图 5：24 小时内缓冲写入与非缓冲写入的 99.9 百分位数延迟性能比较。x 轴上连续刻度线之间的间隔相当于一小时。

- *基于时间戳的对账：*这种情况与前一种情况的不同之处仅在于调和机制。在版本不一致的情况下，Dynamo 会执行简单的基于时间戳的 "最后写入获胜 "调和逻辑；即选择物理时间戳值最大的对象作为正确的版本。维护客户会话信息的服务就是使用这种模式的一个很好的例子。
- *高性能读取引擎：*虽然 Dynamo 是一个 "始终可写 "的数据存储库，但有一些服务正在调整其法定人数特性，并将其用作高性能读取引擎。通常情况下，这些服务的读取请求率很高，更新次数却很少。对于这些服务，Dynamo 提供了在多个节点上分区和复制数据的功能，从而提供了增量可扩展性。其中一些实例可作为权威持久化缓存，用于缓存存储在重量更重的后备存储中的数据。维护产品目录和促销项目的服务就属于此类。

Dynamo 的主要优势在于，客户端应用程序可以调整 N、R 和 W 的值，以达到所需的性能、可用性和耐用性水平。例如，N 值决定了每个对象的耐用性。Dynamo 用户使用的典型 N 值是 3。

W 和 R 的值会影响对象的可用性、耐用性和一致性。例如，如果 W 设置为 1，那么只要系统中至少有一个节点能成功处理写入请求，系统就不会拒绝写入请求。但是，如果 W 和 R 的值较低，则会增加不一致的风险，因为即使大多数副本都没有处理写入请求，系统也会认为写入请求成功并返回给客户端。这还会带来耐久性的漏洞窗口，因为即使写入请求只在少数节点上被持久化，但仍会被成功返回给客户端。

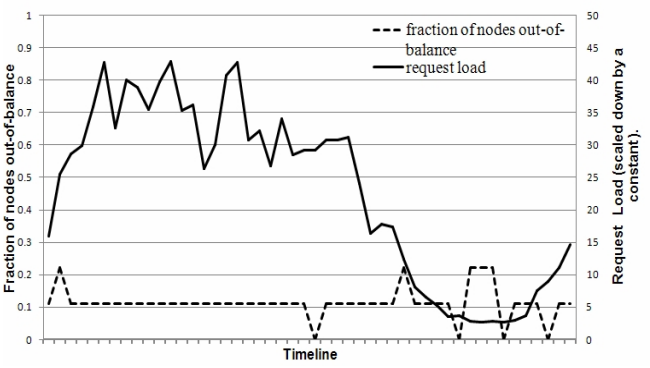


图 6：失衡节点（即请求负载超过系统平均负载某一阈值的节点）的比例及其相应的请求负载。x 轴上的刻度线之间的间隔相当于 30 分钟。

传统观念认为，耐用性和可用性是相辅相成的。然而，这在这里并不一定正确。这可能会增加拒绝请求的概率（从而降低可用性），因为需要更多的存储主机存活才能处理写入请求。

多个 Dynamo 实例常用的 (N,R,W) 配置为 (3,2,2)。选择这些值是为了满足必要的性能、耐用性、一致性和可用性 SLA 水平。

本节介绍的所有测量结果都是在配置为 (3,2,2)、运行着几百个具有相同硬件配置的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接相连。回想一下，要生成成功的获取（或投放）响应，需要 R（或 W）个节点响应协调器。显然，数据中心之间的网络延迟会影响响应时间，而节点（及其数据中心位置）的选择应满足应用的目标 SLA。

6.1 平衡性能与耐用性

虽然 Dynamo 的主要设计目标是建立一个高可用性的数据存储，但性能同样是亚马逊平台的重要标准。如前所述，为了提供一致的客户体验，亚马逊的服务将其性能目标设定为较高的百分位数（如 99.9^{th} 或 99.99^{th} 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 是在 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的，因此为读写操作提供稳定的高性能并非易事。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限于 R 或 W 复制中最慢的一个。图 4 显示了 30 天内 Dynamo 读写操作的平均延迟和 99.9^{th} 百分位数延迟。从图中可以看出，延迟呈现出明显的昼夜变化规律，这是由传入请求率的昼夜变化规律造成的（即，有一个昼夜变化规律）。

白天和夜间的请求率差异很大）。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，99.9th 百分位数的延迟时间约为 200 毫秒，比平均值高出一个数量级。这是因为
99.9th 百分位数延迟受多种因素影响，如请求负载的变化、对象大小和定位模式。

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 提供了权衡性能与耐用性保证的能力。在优化过程中，每个存储节点都会在其主存储器中维护一个对象缓冲区。每个写操作都存储在缓冲区中，并由*写线程*定期写入存储。在此方案中，读取操作首先会检查所请求的密钥是否存在于缓冲区中。如果存在，则从缓冲区而不是存储引擎读取对象。

这一优化将峰值流量期间的 99.9th 百分位数延迟降低了 5 倍，即使缓冲区很小，只有一千个对象（见图 5）。此外，如图所示，写缓冲还能平滑较高的百分位延迟。显然，这种方案是以耐用性换取性能。在这种方案中，服务器崩溃可能会导致缓冲区中排队的写入丢失。为了降低耐久性风险，对写操作进行了改进，让协调者从 N 个副本中选择一个进行 "耐久写"。由于协调器只等待 W 个响应，因此写操作 的性能不会受到单个副本执行的持久写操作性能的影响。

6.2 确保负载分布均匀

Dynamo 使用一致散列法在副本中划分密钥空间，确保负载分布均匀。如果密钥的访问分布不是高度倾斜，均匀的密钥分布可以帮助我们实现均匀的负载分布。特别是，Dynamo 的设计假定，即使访问分布存在明显偏斜，也有足够多的密钥分布在 流行的一端，因此处理流行密钥的负载可以通过分区均匀地分散到各个节点上。本节将讨论 Dynamo 中出现的负载不平衡现象，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，我们测量了每个节点在 24 小时内收到的请求总数--每 30 分钟为一个时间间隔。在给定的时间窗口内，如果节点的请求负载偏离平均负载的值小于某个阈值（此处为 15%），则该节点被视为 "处于平衡状态"。否则，该节点将被视为 "失衡"。图 6 显示了这段时间内 "失衡 "节点的比例（以下简称 " 失衡 率"）。作为参考，图中还显示了整个系统在这段时间内接收到的相应请求负载。从图中可以看出，失衡率随着负载的增加而降低。例如，在低负荷时，不平衡率高达 20%，而在高负荷时则接近 10%。直观地说，这是因为在高负载情况下，大量常用密钥会被访问，而由于密钥分布均匀，负载也就均匀分布了。然而，在低负载时（负载为 1/8th

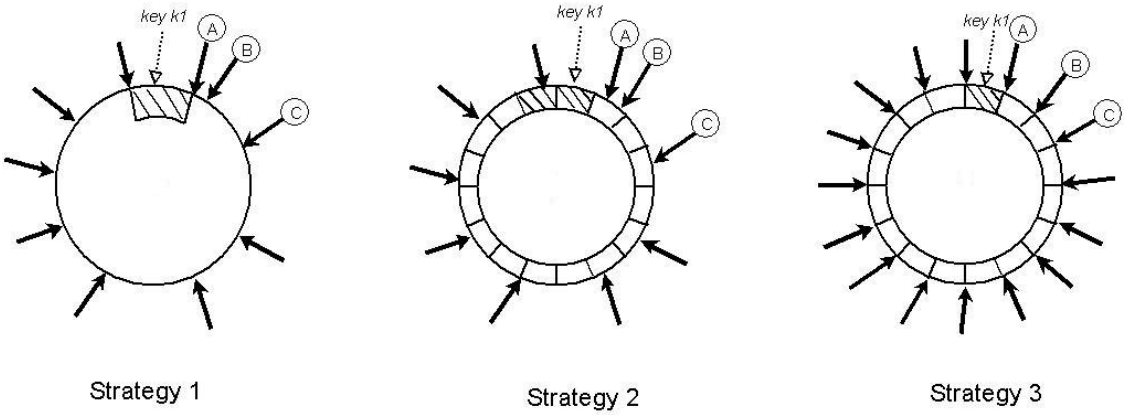


图 7：三种策略中密钥的划分和位置。图中 A、B、C 三个节点构成了一致散列环（N=3）上密钥 k1 的偏好列表。阴影区域表示 A、B 和 C 节点组成首选列表的密钥范围。深色箭头表示不同节点的令牌位置。

的测量峰值负载），访问的常用密钥就会减少，从而导致负载失衡加剧。

本节将讨论 Dynamo 的分区方案随着时间的推移是如何演变的，以及它对负载分布的影响。

策略 1：每个节点 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（在第 4.2 节中进行了描述）。在这种方案中，每个节点都会被分配 T 个令牌（从散列空间中统一随机选择）。所有节点的标记都根据它们在哈希空间中的值排序。每两个连续的标记定义了一个范围。最后一个标记和第一个标记构成一个范围，从散列空间中的最高值 "环绕 " 到最低值。由于标记是随机选择的，因此范围的大小也各不相同。随着节点的加入和离开系统，标记集也会发生变化，范围也会随之改变。请注意，维护每个节点的成员资格所需的空间会随着系统中节点数量的增加而线性增加。

在使用这种策略时，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 "密钥范围。然而，将密钥范围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非常棘手，因为扫描是高度耗费资源的操作，需要在不影响客户性能的情况下在后台执行。这就要求我们以最低优先级运行引导任务。然而，这大大降低了引导过程的速度，在繁忙的购物季节，当节点每天处理数百万个请求时，引导几乎需要一天时间才能完成。其次，当节点加入/离开系统时，许多节点处理的密钥范围都会发生变化，因此需要重新计算新范围的 Merkle 树，这在生产系统上是一项非同小可的操作。最后，由于密钥范围的随机性，没有简单的方法可以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从每个节点分别检索密钥，效率非常低。

这种策略的根本问题在于，数据分区和数据放置方案是相互交织的。例如，在某些情况下，为了处理请求负载的增加，最好在系统中增加更多节点。但在这种情况下，不可能在不影响数据分区的情况下增加节点。理想情况下，最好使用独立的分区和放置方案。为此，对以下策略进行了评估：

*策略 2：每个节点 T 个随机令牌，分区大小相等：*在此策略中，散列空间被划分为 Q 个大小相等的分区/区域，每个节点分配 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。在这种策略中，令牌只是用来建立将哈希空间中的值映射到节点有序列表的函数，而不是决定分区。在从分区末端顺时针方向走一致散列环时，遇到的前 N 个唯一节点上会放置一个分区。图 7 展示了 N=3 时的这一策略。在这个例子中，从包含密钥 k1 的分区末端开始走环，会遇到节点 A、B、C。这种策略的主要优点是(i) 解耦分区和分区放置，以及 (ii) 允许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大小相等的 Q 个分区，分区的位置与分区方案脱钩。此外，每个节点都分配有 Q/S 个代币，其中 S 是系统中的节点数。当一个节点离开系统时，其令牌会随机分配给其余节点，从而保留这些属性。同样，当一个节点加入系统时，它会以保留这些属性的方式从系统中的节点 "偷取 "令牌。

在 S=30 和 N=3 的系统中，对这三种策略的效率进行了评估。然而，要公平地比较这些不同的策 略 并 不 容 易 ， 因为不同的策略有不同的配置来调整其效率。例如，策略 1 的负载分布特性取决于代币数量（即 T），而策略 3 则取决于分区数量（即 Q）。比较这些策略的一个公平方法是

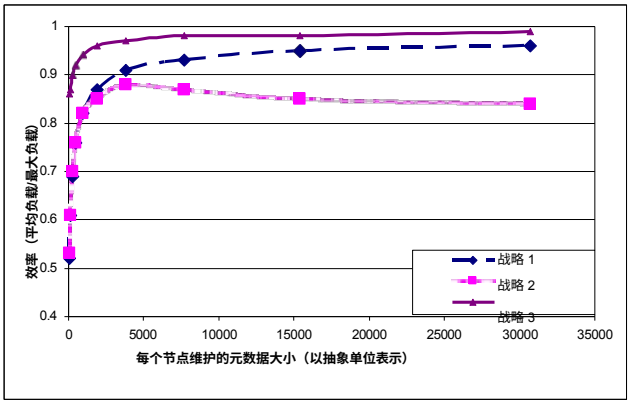


图 8：在 30 个节点和 N=3 的系统中，不同策略的负载分配效率比较，每个节点维护的元数据量相等。系统规模和副本数量的数值是基于我们大多数服务的典型配置。

所有策略都使用相同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标记位置，而在策略 3 中，每个节点都需要维护分配给每个节点的分区信息。

在接下来的实验中，我们通过改变相关参数（T 和 Q）对这些策略进行了评估。在每个节点需要维护的成员信息大小不同的情况下，我们测量了每种策略的负载平衡效率，其中*负载平衡效率*被定义为每个节点提供的平均请求数与最热节点提供的最大请求数之比。

结果见图 8。从图中可以看出，策略 3 实现了最好的负载平衡效率，而策略 2 的负载平衡效率最差。在将 Dynamo 实例从策略 1 迁移到策略 3 的过程中，策略 2 曾短暂充 当临时设置。与策略 1 相比，策略 3 实现了更好的效率，并将每个节点维护的成员信息的大小减少了三个数量级。虽然存储并不是一个大问题，但节点会定期更改成员信息，因此最好尽可能压缩这些信息。除此以外，策略 3 在部署上更具优势，也更简单，原因如下： (i) *启动/恢复更快：*由于分区范围是固定的，它们可以存储在单独的文件中，这意味着只需传输文件就可以将分区作为一个单元重新定位（避免了定位特定项目所需的随机访问）。这简化了启动和恢复过程。(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。在策略 3 中，对 Dynamo 存储的整个数据集进行归档比较简单，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，而且归档存储在 Dynamo 中的数据需要分别从各个节点检索密钥，通常效率低且速度慢。策略 3 的缺点是，改变节点成员资格需要协调，以保持分配所需的属性。

6.3 分歧版本：何时推出？

如前所述，Dynamo 的设计是为了在一致性与可用性。要了解不同故障对一致性的确切影响，需要有关多种因素的详细数据：中断时间、故障类型、组件可靠性、工作量等。详细介绍这些数据超出了本文的讨论范围。不过，本节将讨论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。

数据项的不同版本会在两种情况下出现。第一种情况是系统面临故障，如节点故障、数据中心故障和网络分区。第二种情况是系统要处理大量并发写入单个数据项的情况，多个节点最终要并发协调更新。从可用性和效率的角度来看，最好是在任何给定时间内尽可能减少不同版本的数量。如果仅根据向量时钟无法对版本进行语法调和，就必须将其传递给业务逻辑进行语义调和。语义调和会给服务带来额外负担，因此最好尽量减少这种需要。

在下一个实验中，我们对 24 小时内返回购物车服务的版本数量进行了分析。在此期间，99.94% 的请求只看到一个版本；0.00057% 的请求看到 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。这表明很少会出现不同版本。

经验表明，不同版本数量的增加不是由于故障，而是由于并发写入数量的增加。并发写入数量的增加通常是由繁忙的机器人（自动客户端程序）触发的，很少是由人类触发的。由于故事的敏感性，我们没有详细讨论这个问题。

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偏好列表中的节点协调。之所以有这样的限制，是因为这些首选节点还要负责创建一个新的版本戳，该版本戳会因果地覆盖写请求更新的版本。请注意，如果 Dynamo 的版本控制方案基于物理时间戳，那么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至客户端节点。在这种方案中，客户端应用程序使用库在本地执行请求协调。客户端会定期随机选择一个 Dynamo 节点，并下载其当前的 Dynamo 成员状态视图。利用这些信息，客户端可以确定哪一组节点构成了任何给定密钥的首选列表。读取请求可以在客户端节点进行协调，从而避免了额外的网络跳转，如果请求是由负载平衡器随机分配给一个 Dynamo 节点，则会产生额外的网络跳转。写入请求要么会被转发到密钥首选列表中的节点，要么会被转发到密钥首选列表中的节点。

表 2：客户端驱动和服务器驱动协调方法的性能。

	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管理，则在本地进行协调。

客户端驱动协调方法的一个重要优势是，不再需要负载平衡器来统一分配客户端负载。将密钥近乎统一地分配给存储节点，就能隐含地保证公平的负载分配。显然，这种方案的效率取决于客户端成员信息的新鲜程度。目前，客户端每 10 秒就会随机轮询一个 Dynamo 节点，以获取成员信息更新。之所以选择基于拉动的方法而不是基于推送的方法，是因为前者在客户端数量较多的情况下扩展性更好，而且只需要在服务器上维护很少的客户端状态。不过，在最坏的情况下，客户端可能会在 10 秒钟的时间内暴露在陈旧的成员信息中。在这种情况下，如果客户端检测到其成员表是陈旧的（例如，当一些成员无法访问时），它会立即刷新其成员信息。

表 2 显示了在 24 小时内使用客户端驱动协调方法与服务器驱动方法相比，在 99.9^{th} 百分位数和平均值上观察到的延迟改善情况。从表中可以看出，客户端驱动的协调方法在以下情况下至少减少了 30 毫秒的延迟
99.9^{th} 百分位数的延迟，并将平均延迟缩短了 3 至 4 毫秒。延迟改善的原因是，客户端驱动方法消除了负载平衡器的开销，以及将请求分配给随机节点时可能产生的额外网络跳转。从表中可以看出，平均延迟往往明显低于 99.9 百分位数的延迟。这是因为 Dynamo 的存储引擎缓存和写缓冲区具有良好的命中率。此外，由于负载平衡器和网络会给响应时间带来额外的变化，因此 99.9^{th} 百分位数的响应时间增益要高于平均值。

6.5 平衡后台与前台任务

除了正常的前台put/get操作外，每个节点都会执行不同类型的后台任务，用于复制同步和数据移交（提示或添加/删除节点）。在早期的生产环境中，这些后台任务引发了资源争用问题，并影响了 常规put 和 get 操作的性能。因此，有必要确保后台任务只在常规关键操作不会受到严重影响的情况下运行。为 此，我们将后台任务与准入控制机制整合在一起。每个后台任务都使用该控制器来保留资源（如数据库）的运行时片段、

所有后台任务共享。根据监测到的前台任务性能，采用反馈机制来改变后台任务可用的切片数量。

在执行 "前台 "put/get 操作时，准入控制器会持续监控资源访问行为。监控的方面包括磁盘操作的延迟、因锁保留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。这些信息用于检查给定尾随时间窗口中的延迟（或故障）百分位数是否接近所需的阈值。例如，后台控制器会检查 99^{th} 百分位数数据库读取延迟（过去 60 秒）与预设阈值（如 50 毫秒）的接近程度。控制器利用这种比较来评估前台操作的资源可用性。随后，它将决定有多少时间片可用于后台任务，从而利用反馈回路限制后台活动的干扰性。请注意，类似的后台任务管理问题在 [4] 中也有研究。

6.6 讨论

本节总结了在实施和维护 Dynamo 过程中获得的一些经验。在过去两年中，许多亚马逊内部服务都使用了 Dynamo，它为其应用程序提供了显著的可用性。特别是，应用程序 99.9995% 的请求都得到了成功响应（没有超时），迄今为止没有发生过数据丢失事件。

此外，Dynamo 的主要优势还在于它提供了必要的旋钮，可使用三个参数（N、R、W）根据需要调整实例。与流行的商业数据存储不同，Dynamo 将数据一致性和调节逻辑问题暴露给开发人员。一开始，人们可能会认为应用逻辑会变得更加复杂。但是，从历史上看，亚马逊的平台是为高可用性而构建的，许多应用程序都是为处理可能出现的不同故障模式和不一致性而设计的。因此，将此类应用程序移植到 Dynamo 是一项相对简单的任务。对于希望使用 Dynamo 的新应用程序，需要在开发的初始阶段进行一些分析，以选择适当的冲突解决机制，满足业务需求。最后，Dynamo 采用完全成员模型，即每个节点都知道其对等节点托管的数据。为此，每个节点都会主动向系统中的其他节点透露完整的路由表。这种模式在包含数百个节点的系统中运行良好。然而，要将这种设计扩展到数万个节点的运行规模却并非易事，因为维护路由表的开销会随着系统规模的扩大而增加。可以通过对 Dynamo 进行分层扩展来克服这一限制。此外，O(1) DHT 系统（如 [14]）也在积极解决这一问题。

7. 结论

本文介绍了 Dynamo，这是一个高可用性和可扩展的数据存储库，用于存储亚马逊公司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，并成功处理了服务器故障、数据中心故障和网络分区。Dynamo 可逐步扩展，允许服务所有者根据其当前的需求进行增减。

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

在过去一年中，Dynamo 的生产使用表明，分散式技术可以结合起来提供单一的高可用性系统。它在最具挑战性的应用环境中取得的成功表明，最终一致的存储系统可以成为高可用性应用的基石。

致谢

作者感谢 Pat Helland 对 Dynamo 最初设计的贡献。我们还要感谢 Marvin Theimer 和 Robert van Renesse 的意见。最后，我们要感谢我们的指导老师杰夫-莫格（Jeff Mogul），感谢他在准备照相版本时提出的详细意见和建议，这些意见和建议大大提高了论文的质量。

参考文献

[1] Adya, A., Bolosky, W. J., Castro, M., Cermak, G., Chaiken、R., Douceur, J. R., Howell, J., Lorch, J. R., Theimer, M., and Wattenhofer, R. P. 2002.Farsite: federated, available, and reliable storage for an incompletely trusted environment. *SIGOPS Oper.Syst.Rev.* 36, SI (Dec. 2002), 1-14.

[2] Bernstein, P.A., and Goodman, N. An algorithm for concurrency control and recovery in replicated distributed databases.ACM Trans. on Database Systems, 9(4):596-615
、
1984 年 12 月

[3] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach、D.A.、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。In *Proceedings of the 7th Conference on USENIX Symposium on Operating Systems Design and Implementation - Volume 7* (Seattle, WA, November 06 - 08, 2006).USENIX 协会，加利福尼亚州伯克利，15-15。

[4] Douceur, J. R. and Bolosky, W. J. 2000.基于流程的低重要性流程监管。*SIGOPS Oper.Syst.*34, 2 (Apr. 2000), 26-27.

[5] Fox, A., Gribble, S. D., Chawathe, Y., Brewer, E. A., and Gauthier, P. 1997.基于集群的可扩展网络服务。*第十六届 ACM 操作系统原理研讨会论文集*》（法国圣马洛，1997 年 10 月 05 - 08 日）。W. M. Waite, Ed.SOSP '97.ACM Press, New York, NY, 78-91.

[6] Ghemawat, S., Gobioff, H., and Leung, S. 2003.谷歌文件系统*第十九届 ACM 操作系统原理研讨会论文集*》（美国纽约博尔顿兰丁，2003 年 10 月 19-22 日）。SOSP '03.ACM
Press, New York, NY, 29-43.

[7] Gray, J., Helland, P., O'Neil, P., and Shasha, D. 1996.复制的危险与解决方案。In *Proceedings of the 1996 ACM SIGMOD international Conference on Management of Data* (Montreal, Quebec, Canada, June 04 - 06, 1996).J. Widom, Ed.SIGMOD '96.ACM Press, New York, NY, 173-182.

[8] Gupta, I., Chandra, T. D., and Goldszmidt, G. S. 2001.关于可扩展的高效分布式故障检测器。In *Proceedings of the Twentieth Annual ACM Symposium on*

分布式计算原理（美国罗德岛新港）。PODC '01.ACM Press, New York, NY, 170-179.

[9] Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Wells, C., and Zhao, B. 2000.OceanStore: an architecture for global-scale persistent storage.*SIGARCH Comput.Archit.新闻* 28, 5 (Dec. 2000), 190-201.

[10] Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., and Lewin, D. 1997.一致散列与随机树：缓解万维网热点的分布式缓存协议》。In *Proceedings of the Twenty-Ninth Annual ACM Symposium on theory of Computing* (El Paso, Texas, United States, May 04 - 06, 1997).STOC '97.ACM Press, New York, NY, 654-663.

[11] Lindsay, B.G., et. al., "Notes on Distributed Databases"、研究报告 RJ2571（33471），IBM 研究部，1979 年 7 月

[12] Lamport, L. Time, clocks, and the ordering of events in a distributed system.ACM Communications, 21(7), pp.

[13] Merkle, R. 基于传统加密函数的数字签名。CRYPTO 会议论文集，第 369- 页。378.施普林格出版社，1988 年。

[14] Ramasubramanian, V., and Sirer, E. G. Beehive：点对点重叠中幂律查询分布的 O(1)lookup 性能。*第一届网络系统设计与实施研讨会论文集》*，加利福尼亚州旧金山，2004 年 3 月 29-31 日。

[15] Reiher, P., Heidemann, J., Ratner, D., Skinner, G., and Popek, G. 1994.解决 Ficus 文件系统中的文件冲突。In *Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference - Volume 1* (Boston, Massachusetts, June 06 - 10, 1994).USENIX 协会，加利福尼亚州伯克利，12-12 。

[16] Rowstron, A., and Druschel, P. Pastry：大规模点对点系统的可扩展、分散式对象定位与路由。中间件论文集》，第 329-350 页，2001 年 11 月。

[17] Rowstron, A., and Druschel, P. Storage management and caching in PAST, a large-scale, persistent peer-to-peer storage utility.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., and Spence 、S. 2004.FAB: Building distributed enterprise disk arrays from commodity components.*SIGOPS Oper.Syst.Rev.* 38, 5 (Dec. 2004), 48-58.

[19] Satyanarayanan, M., Kistler, J.J., Siegel, E.H. Coda: A Resilient Distributed File System.电气和电子工程师学会工作站操作系统研讨会，1987 年 11 月。

[20] Stoica, I., Morris, R., Karger, D., Kaashoek, M. F., and Balakrishnan, H. 2001.Chord：互联网应用的可扩展点对点查询服务。2001 年 *计算机通信应用、技术、架构和协议会议论文集*（美国加利福尼亚州圣迭戈）。SIGCOMM '01.ACM Press, New York, NY, 149-160.

[21] Terry, D. B., Theimer, M. M., Petersen, K., Demers, A. J., Spreitzer, M. J., and Hauser, C. H. 1995.在弱连接复制存储系统 Bayou 中管理更新冲突。*第 15 届 ACM 操作系统原理研讨会论文集*（美国科罗拉多州铜山，1995 年 12 月 3-6 日）。M. B. Jones, Ed.SOSP '95.ACM Press, New York, NY, 172-182.

[22] Thomas, R. H. A majority consensus approach to concurrency control for multiple copy databases.ACM Transactions on Database Systems 4 (2)：180-209, 1979.

[23] Weatherspoon, H., Eaton, P., Chun, B., and Kubiatowicz, J. 2007.Antiquity: exploiting a secure log for wide-area distributed storage.*SIGOPS Oper.Syst.*41, 3 (Jun. 2007), 371-384.

[24] Welsh, M., Culler, D., and Brewer, E. 2001.SEDA: an architecture for well-conditioned, scalable internet services. *第十八届 ACM 操作系统原理研讨会论文集*》（加拿大阿尔伯塔省班夫，2001 年 10 月 21-24 日）。SOSP '01.ACM Press, New York、纽约，230-243。