Dynamo：Amazon 的高可用性的键-值存储系统

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

摘要

巨大规模系统的可靠性是我们在 Amazon.com，这个世界上最大的电子商务公司之一，面对最大的挑战之一，即使最轻微的系统中断都有显着的经济后果并且影响到客户的信赖。Amazon.com 平台，它为全球许多网站服务，是实现在位于世界各地的许多数据中心中的成千上万的服务器和网络基础设施之上。在这一规模中，各种大大小小的部件故障持续不断发生，管理持久化状态的方法在面对这些故障时，驱使软件系统的可靠性和可扩展性。

本文介绍 Dynamo 的设计和实现，一个高度可用的 key-value 存储系统，一些 Amazon 的核心服务使用它用以提供一个“永远在线”的用户体验。为了达到这个级别的可用性，Dynamo 在某些故障的场景中将牺牲一致性。它大量使用对象版本和应用程序协助的冲突协调方式以提供一个开发人员可以使用的新颖接口。

1 简介

Amazon 运行一个全球性的电子商务服务平台，在繁忙时段使用位于世界各地的许多数据中心的数千台服务器为几千万的客户服务。Amazon 平台有严格的性能，可靠性和效率方面操作要求，并支持持续增长，因此平台需要高度可扩展性。可靠性是最重要的要求之一，因为即使最轻微的系统中断都有显著的经济后果和影响客户的信赖。此外，为了支持持续增长，平台需要高度可扩展性。

我们组织在运营 Amazon 平台时所获得的教训之一是，一个系统的可靠性和可扩展性依赖于它的应用状态如何管理。Amazon 采用一种高度去中心化(decentralized)，松散耦合，由数百个服务组成的面向服务架构。在这种环境中特别需要一个始终可用存储技术。例如，即使磁盘故障，网络路线状态摇摆不定，或数据中心被龙卷风摧毁时，客户应该能够查看和添加物品到自己的购物车。因此，负责管理购物车的服务，它可以随时写入和读取数据，并且数据需要跨越多个数据中心。

在一个由数百万个组件组成的基础设施中进行故障处理是我们的标准运作模式;在任何给定的时间段，总有一些小的但相当数量的服务器和网络组件故障，因此 Amazon 的软件系统需要将错误处理当作正常情况下来建造，而不影响可用性或性能。

为了满足可靠性和可伸缩性的需要，Amazon 开发了许多存储技术，其中 Amazon 简单存储服务(一个 Amazon 之外也可获得的服务,即广为人知的 AmazonS3 - SimpleStorageService)大概是最为人熟知。本文介绍了 Dynamo 的设计和实现，另一个为 Amazon 的平台上构建的高度可用和可扩展的分布式数据存储系统。Dynamo 被用来管理服务状态并且要求非常高的可靠性，而且需要严格控制可用性，一致性，成本效益和性能的之间的权衡。Amazon 平台的不同应用对存储要求非常差异非常高。一部分应用需要存储技术具有足够的灵活性，让应用程序设计人员配置适当的数据存储来达到一种平衡，以实现高可用性和最具成本效益的方式保证性能。

Amazon 服务平台中的许多服务只需要主键访问数据存储。对于许多服务，如提供最畅销书排行榜，购物车，客户的偏好，会话管理，销售等级，产品目录，常见的使用关系数据库的模式会导致效率低下，有限的可扩展性和可用性。Dynamo 提供了一个简单的主键唯一的接口，以满足这些应用的要求。

Dynamo 综合了一些著名的技术来实现可伸缩性和可用性：数据划分(data partitioned)和使用一致性哈希的复制(replicated)[10]，并通过对象版本(object versioning)提供一致性[12]。在更新时，副本之间的一致性是由仲裁般(quorum-like)的技术和去中心化的副本同步协议来维持的。Dynamo 采用了基于 gossip(不知道怎样译过来好，流言蜚语?或许保留成外来语为好)的分布式故障检测及成员(membership)协议(也即 token 环上的节点在响应节点加入(join)/离开(leaving)/移除(removing)/消亡(dead)等所采取的动作以维持 DHT/Partitioning 的正确语义)。Dynamo 是一个只需要很少的人工管理，去中心化的系统。存储节点可以添加和删除，而不需要任何手动划分(partitioning - partitioner controls how the data are distributed over the nodes)或重新分配(redistribution)。

在过去的一年，Dynamo 已经成为 Amazon 电子商务平台的核心服务的底层存储技术。它能够有效地扩展到极端高峰负载，在繁忙的假日购物季节没有任何的停机时间。例如，维护购物车(购物车服务)的服务，在一天内承担数千万的请求，并因此导致超过 300 万 checkouts(结算？)，以及管理十万计的并发活动会话的状态。

这项工作对研究社区的主要贡献是评估不同的技术如何能够结合在一起，并最终提供一个单一的高可用性的系统。它表明一个最终一致(eventually-consistent)的存储系统可以在生产环境中被苛刻的应用程序所采用。它也对这些技术的调整的进行深入的分析，以满足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 节背景知识，第 3 节阐述有关工作，第 4 节介绍了系统设计，第 5 描述了实现，第 6 条详述经验和在生产运营 Dynamo 时取得的经理，第 7 节结束本文。在这其中，有些地方也许可以适当有更多的信息，但出于适当保护 Amazon 的商业利益，要求我们在文中降低详细程度。由于这个原因，第 6 节数据中心内和跨数据中心之间的延时，第 6.2 节的相对请求速率，以及第 6.3 节系统中断(outage)的时长，系统负载，都只是总体测量而不是绝对的详细。

2 背景

Amazon 电子商务平台由数百个服务组成，它们协同工作，提供的功能包括建议，完成订单到欺诈检测。每个服务是通过一个明确的公开接口，通过网络访问。这些服务运行在位于世界各地的许多数据中心的数万台服务器组成的基础设施之上。其中一些服务是无状态(比如，一些服务只是收集(aggregate)其他服务的 response)，有些是有状态的(比如，通过使用持久性存储区中的状态，，执行业务逻辑，进而生成 response)。

传统的生产系统将状态存储在关系数据库中。对于许多更通用的状态存储模式，关系数据库方案是远不够理想。因为这些服务大多只通过数据的主键存储和检索数据，并且不需要 RDBMS 提供的复杂的查询和管理功能。这多余的功能，其运作需要昂贵的硬件和高技能人才，使其成为一个非常低效的解决方案。此外，现有的复制(replication)技术是有限的，通常选择一致性是以牺牲可用性为代价(typically choose consistency over availability)。虽然最近几年已经提出了许多进展，但数据库水平扩展(scaleout)或使用负载平衡智能划分方案仍然不那么容易。

本文介绍了 Dynamo，一个高度可用的数据存储技术，能够满足这些重要类型的服务的需求。Dynamo 有一个简单的键/值接口，它是高可用的并同时具有清晰定义的一致性滑动窗口，它在资源利用方面是高效的，并且在解决规模增长或请求率上升时具有一个简单的水平扩展(scaleout)方案。每个使用 Dynamo 的服务运行它自己的 Dynamo 实例。

2.1 系统假设和要求

这种类型的服务的存储系统具有以下要求：

查询模型：对数据项简单的读，写是通过一个主键唯一性标识。状态存储为一个由唯一性键确定二进制对象(即 blob，呵呵，java 就是 ByteBuffer 啦)。没有横跨多个数据项的操作，也不需要关系方案(relational schema)。这项规定是基于观察，相当一部分 Amazon 的服务可以使用这个简单的查询模型，并不需要任何关系模式。Dynamo 的目标应用程序需要存储的对象都比较小(通常小于 1MB)。

ACID 属性：ACID(*原子性，一致性，隔离性，持久性*)是一种保证数据库事务可靠地处理的属性。在数据库方面的，对数据的单一的逻辑操作被称作所谓的交易。Amazon 的经验表明，在保证 ACID 的数据存储提往往有很差的可用性。这已被业界和学术界所公认[5]。Dynamo 的目标应用程序是高可用性，弱一致性(ACID 中的“C”)。Dynamo 不提供任何数据隔离(Isolation)保证，只允许单一的关键更新。

效率：系统需运作在“日用品”(commodity，非常喜欢这个词，因为可以在家做试验！)级的硬件基础设施上。Amazon 平台的服务都有着严格的延时要求，一般延时所需要度量到分布的 99.9 百分位。在服务操作中鉴于对状态的访问起着至关重要的作用，存储系统必须能够满足那些严格的 SLA(见以下 2.2)，服务必须能够通过配置 Dynamo，使他们不断达到延时和吞吐量的要求。因此，必须在成本效率，可用性和耐用性保证之间做权衡。

其他假设：Dynamo 仅被 Amazon 内部的服务使用。它的操作环境被假定为不怀恶意的(non-hostile)，没有任何安全相关的身份验证和授权的要求。此外，由于每个服务使用其特定的 Dynamo 实例，它的最初设计目标的规模高达上百的存储主机。我们将在后面的章节讨论 Dynamo 可扩展性的限制和相关可能的扩展性的延伸。

2.2 服务水平协议(SLA)

为了保证应用程序可以在限定的(bounded)时间内递送(deliver)其功能，一个平台内的任何一个依赖都在一个更加限定的时间内递送其功能。客户端和服务端采用服务水平协议(SLA)，其为客户端和服务端在几个系统相关的特征上达成一致的一个正式协商合约，其中，最突出的包括客户对特定的 API 的请求速率分布的预期要求，以及根据这些条件，服务的预期延时。一个简单的例子是一个服务的 SLA 保证：在客户端每秒 500 个请求负载高峰时，99.9%的响应时间为 300 毫秒。

在 Amazon 的去中心化的面向服务的基础设施中，服务水平协议发挥了重要作用。例如，一个页面请求某个电子商务网站，通常需要页面渲染(rendering)引擎通过发送请求到 150 多个服务来构造其响应。这些服务通常有多个依赖关系，这往往是其他服务，因此，有一层以上调用路径的应用程序通常并不少见。为了确保该网页渲染引擎在递送页面时可以保持明确的时限，调用链内的每个服务必须履行合约中的性能指标。

图 1 显示了 Amazon 平台的抽象架构，动态网页的内容是由页面呈现组件生成，该组件进而查询许多其他服务。一个服务可以使用不同的数据存储来管理其状态，这些数据存储仅在其服务范围才能访问。有些服务作为聚合器使用其他一些服务，可产生合成(composite)响应。通常情况下，聚合服务是无状态，虽然他们利用广泛的缓存。

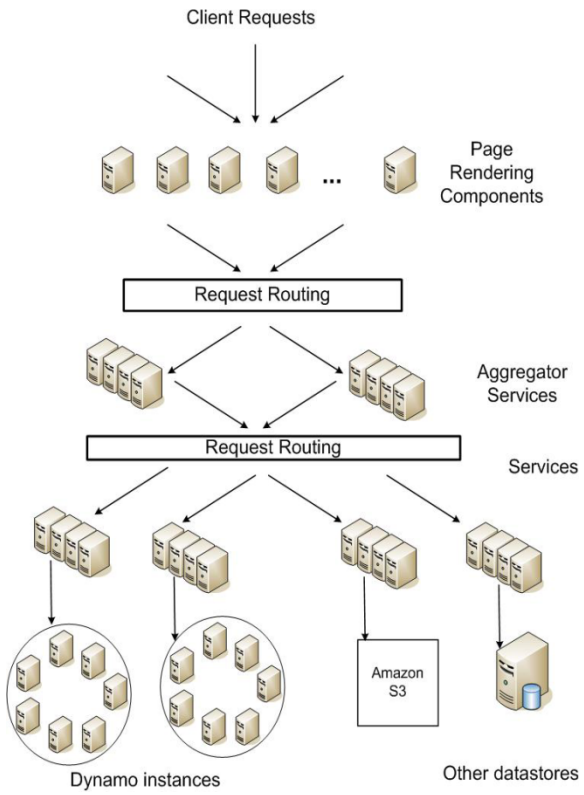

图 1：面向服务的 Amazon 平台架构。

行业中，表示面向性能的 SLA 的共同做法是使用平均数(average)，中值(median)和预期变化(expected variance)。在 Amazon，我们发现，这些指标不够好，如果目标是建立一个对**所有**，而不是大多数客户都有着良好体验的系统。例如，如果个性化(personalization)技术被广泛使用，那么有很长的历史的客户需要更多的处理，性能影响将表现在分布的高端。前面所述的基于平均或中值响应时间的 SLA 不能解决这一重要客户段的性能问题。为了解决这个问题，在 Amazon，SLA 是基于分布的 99.9 百分位来表达和测量的。选择百分位 99.9 的而不是更高是根据成本效益分析，其显示出在 99.9 之后，要继续提高性能，成本将大幅增加。系统的经验与 Amazon 的生产表明，相比于那些基于平均或中值定义的 SLA 的系统，该方法提供了更好的整体体验。

本文多次提到这种 99.9 百分位分布，这反映了 Amazon 工程师从客户体验角度对性能不懈追求。许多论文统计平均数，所以在本论文的一些地方包括它可以用来作比较。然而，Amazon 的工程和优化没有侧重于平均数。几种技术，如作为写协调器(coordinators)的负载均衡的选择，纯粹是针对控制性能在 99.9 百分位的。

存储系统在建立一个服务的 SLA 中通常扮演重要角色，特别是如果业务逻辑是比较轻量级时，正如许多 Amazon 的服务的情况。状态管理就成为一个服务的 SLA 的主要组成部分。对 dynamo 的主要设计考虑的问题之一就是给各个服务控制权，通过系统属性来控制其耐用性和一致性，并让服务自己在功能，性能和成本效益之间进行权衡。

2.3 设计考虑

在商业系统中，数据复制(Data replication)算法传统上执行同步的副本(replica)协调，以提供一个强一致性的数据访问接口。为了达到这个水平的一致性，在某些故障情况下，这些算法被迫牺牲了数据可用性。例如，与其不能确定答案的正确性与否，不如让该数据一直不可用直到它绝对正确时。

从最早期的备份(replicated)数据库，众所周知，当网络故障时，强一致性和高可用性不可能性同时实现[2，11]。因此，系统和应用程序需要知道在何种情况下可以达到哪些属性。

对于容易出现的服务器和网络故障的系统，可使用乐观复制技术来提高系统的可用性，其变化可以在后台传播到副本，同时，并发和断开(disconnected)是可以容忍的。这种方法的挑战在于，它会导致更改冲突，而这些冲突必须检测并协调解决。这种协调冲突的过程引入了两个问题：何时协调它们，谁协调它们。Dynamo 被设计成最终一致性(eventually consistent)的数据存储，即所有的更新操作，最终达到所有副本。

一个重要的设计考虑的因素是决定*何时*去协调更新操作冲突，即是否应该在读或写过程中协调冲突。许多传统数据存储在写的过程中执行协调冲突过程，从而保持读的复杂度相对简单[7]。在这种系统中，如果在给定的时间内数据存储不能达到所要求的所有或大多数副本数，写入可能会被拒绝。另一方面，Dynamo 的目标是一个“永远可写”(always writable)的数据存储(即数据存储的“写”是高可用)。对于 Amazon 许多服务来讲，拒绝客户的更新操作可能导致糟糕的客户体验。例如，即使服务器或网络故障，购物车服务必须让客户仍然可向他们的购物车中添加和删除项。这项规定迫使我们将协调冲突的复杂性推给“读”，以确保“写”永远不会拒绝。

下一设计选择是*谁*执行协调冲突的过程。这可以通过数据存储或客户应用程序。如果冲突的协调是通过数据存储，它的选择是相当有限的。在这种情况下，数据存储只可能使用简单的策略，如”最后一次写入获胜“(last write wins)[22]，以协调冲突的更新操作。另一方面，客户应用程序，因为应用知道数据方案，因此它可以基于最适合的客户体验来决定协调冲突的方法。例如，维护客户的购物车的应用程序，可以选择“合并”冲突的版本，并返回一个统一的购物车。尽管具有这种灵活性，某些应用程序开发人员可能不希望写自己的协调冲突的机制，并选择下压到数据存储，从而选择简单的策略，例如“最后一次写入获胜”。

设计中包含的其他重要的设计原则是：

增量的可扩展性：Dynamo 应能够一次水平扩展一台存储主机(以下，简称为“*节点*“)，而对系统操作者和系统本身的影响很小。

对称性：每个 Dynamo 节点应该与它的对等节点(peers)有一样的责任;不应该存在有区别的节点或采取特殊的角色或额外的责任的节。根据我们的经验，对称性(symmetry)简化了系统的配置和维护。

去中心化：是对对称性的延伸，设计应采用有利于去中心化而不是集中控制的技术。在过去，集中控制的设计造成系统中断(outages)，而本目标是尽可能避免它。这最终造就一个更简单，更具扩展性，更可用的系统。

异质性：系统必须能够利用异质性的基础设施运行。例如，负载的分配必须与各个独立的服务器的能力成比例。这样就可以一次只增加一个高处理能力的节点，而无需一次升级所有的主机。

3 相关工作

3.1 点对点系统

已经有几个点对点(P2P)系统研究过数据存储和分配的问题。如第一代P2P系统Freenet和Gnutella，被主要用作文件共享系统。这些都是由链路任意建立的非结构化 P2P 的网络的例子。在这些网络中，通常是充斥着通过网络的搜索查询以找到尽可能多地共享数据的对等节点。P2P 系统演进到下一代是广泛被称为结构化 P2P。这些网络采用了全局一致的协议，以确保任何节点都可以有效率地传送一个搜索查询到那些需要数据的节点。系统，如 Pastry[16]和 Chord[20]使用路由机制，以确保查询可以在有限的跳数(hops)内得到回答。为了减少多跳路由引入的额外延时，有些 P2P 系统(例如，[14])采用 O(1)路由，每个节点保持足够的路由信息，以便它可以在常数跳数下从本地路由请求(到要访问的数据项)到适当的节点。

其他的存储系统，如 Oceanstore[9]和 PAST[17]是建立在这些交错的路由基础之上的。Oceanstore 提供了一个全局性的，事务性，持久性存储服务，支持对广阔的(widely)复制的数据进行序列化更新。为允许并发更新的同时避免与广域锁定(wide-area locking)固有的许多问题，它使用了一个协调冲突的更新模型。[21]介绍了在协调冲突，以减少交易中止数量。Oceanstore 协调冲突的方式是：通过处理一系列的更新，为他们选择一个最终的顺序，然后利用这个顺序原子地进行更新。它是为数据被复制到不受信任的环境的基础设施之上而建立。相比之下，PAST 提供了一个基于 Pastry 和不可改变的对象的简单的抽象层。它假定应用程序可以在它之上建立必要的存储的语义(如可变文件)。

3.2 分布式文件系统和数据库

出于对性能，可用性和耐用性考虑，数据分发已被文件系统和数据库系统的社区广泛研究。对比于 P2P 存储系统的只支持平展的命名空间，分布式文件系统通常支持分层的命名空间。系统象 Ficus[15]和 Coda[19]其文件复制是以牺牲一致性为代价而达到高可用性。更新冲突管理通常使用专门的协调冲突程序。Farsite 系统[1]是一个分布式文件系统，不使用任何类似 NFS 的中心服务器。Farsite 使用复制来实现高可用性和可扩展性。谷歌文件系统[6]是另一个分布式文件系统用来承载谷歌的内部应用程序的状态。GFS 使用简单的设计并采用单一的中心(master)服务器管理整个元数据，并且将数据被分成块存储在 chunkservers 上。Bayou 是一个分布式关系数据库系统允许断开(disconnected)操作，并提供最终的数据一致性[21]。

在这些系统中，Bayou,Coda 和 Ficus 允许断开操作和有从如网络分裂和中断的问题中复原的弹性。这些系统的不同之处在于协调冲突程序。例如，Coda 和 Ficus 执行系统级协调冲突方案，Bayou 允许应用程序级的解决方案。不过，所有这些都保证最终一致性。类似这些系统，Dynamo 允许甚至在网络被分裂(partition - *network partition*, which is a break in the network that prevents one machine in one data center from interacting directly with another machine in other data center)的情况下继续进行读，写操作，以及使用不同的机制来协调有冲突的更新操作。分布式块存储系统，像 FAB[18]将大对象分成较小的块，并以高度可用的方式存储。相对于这些系统中，一个 key-value 存储在这种情况下更合适，因为：(a)它就是为了存放相对小的物体(大小<1M)和 (b)key-value 存储是以每个应用更容易配置为基础的。Antiquity 是一个广域分布式存储系统，专为处理多种服务器故障[23]。它使用一个安全的日志来保持数据的完整性，复制日志到多个服务器以达到耐久性，并使用 Byzantine 容错协议来保证数据的一致性。相对于 antiquity，Dynamo 不太注重数据完整性和安全问题，并为一个可信赖的环境而建立的。BigTable 是一个管理结构化数据的分布式存储系统。它保留着稀疏的,多维的有序映射，并允许应用程序使用多个属性访问他们的数据[2]。相对于 Bigtable 中，Dynamo 的目标应用程序只需要 key/value 并主要关注高可用性，甚至在网络分裂或服务器故障时，更新操作都不会被拒绝。

传统备份(replicated)关系数据库系统强调保证复制数据的一致性。虽然强一致性给应用编写者提供了一个更方便的应用程序编程模型，但这些系统都只有有限的可伸缩性和可用性[7]。这些系统不能处理网络分裂(partition)，因为它们通常提供强的一致性保证。

3.3 讨论

与上述去中心化的存储系统相比，Dynamo 有着不同的目标需求：首先，Dynamo 主要是针对应用程序需要一个“永远可写”数据存储，不会由于故障或并发写入而导致更新操作被拒绝。这是许多 Amazon 应用的关键要求。其次，如前所述，Dynamo 是建立在一个所有节点被认为是值得信赖的单个管理域的基础设施之上。第三，使用 Dynamo 的应用程序不需要支持分层命名空间(许多文件系统采用的规范)或复杂的(由传统的数据库支持)关系模式的支持。第四，Dynamo 是为延时敏感应用程序设计的，需要至少 99.9%的读取和写入操作必须在几百毫秒内完成。为了满足这些严格的延时要求，这促使我们必须避免通过多个节点路由请求(这是被多个分布式哈希系统如 Chord 和 Pastry 采用典型的设计)。这是因为多跳路由将增加响应时间的可变性，从而导致百分较高的延时的增加。Dynamo 可以被定性为零跳(zero-hop)的 DHT，每个节点维护足够的路由信息从而直接从本地将请求路由到相应的节点。

4 系统架构

一个操作在生产环境里的存储系统的架构是复杂的。除了实际的数据持久化组件，系统需要有负载平衡，成员(membership)和故障检测，故障恢复，副本同步，过载处理，状态转移，并发性和工作调度，请求 marshaling，请求路由，系统监控和报警，以及配置管理等可扩展的且强大的解决方案。描述解决方案的每一个细节是不可能的，因此本文的重点是核心技术在分布式系统中使用 Dynamo：划分(partitioning)，复制(replication)，版本(versioning)，会员(membership)，故障处理(failure handling)和伸缩性(scaling)。表 1 给出了简要的 Dynamo 使用的技术清单和各自的优势。

表 1：Dynamo 使用的技术概要和其优势。

问题	技术	优势
划分	一致性哈希	增量可伸缩性
写的高可用性	矢量时钟与读取过程中的协调(reconciliation)	版本大小与更新操作速率脱钩。
暂时性的失败处理	草率仲裁(Sloppy Quorum?)，并暗示移交(hinted handoff)	提供高可用性和耐用性的保证，即使一些副本不可用时。
永久故障恢复	使用 Merkle 树的反熵(Anti-entropy)	在后台同步不同的副本。
会员和故障检测	Gossip 的成员和故障检测协议。	保持对称性并且避免了一个用于存储会员和节点活性信息的集中注册服务节点。

4.1 系统接口

Dynamo 通过一个简单的接口将对象与 key 关联，它暴露了两个操作：get()和 put()。get(key)操作在存储系统中定位与 key 关联的对象副本，并返回一个对象或一个包含冲突的版本和对应的上下文对象列表。put(key,context,object)操作基于关联的 key 决定将对象的副本放在哪，并将副本写入到

磁盘。该 *context* 包含对象的系统元数据并对于调用者是不透明的(opaque)，并且包括如对象的版本信息。上下文信息是与对象一起存储，以便系统可以验证请求中提供的上下文的有效性。

Dynamo 将调用者提供的 key 和对象当成一个不透明的字节数组。它使用 MD5 对 key 进行 Hash 以产生一个 128 位的标识符，它是用来确定负责(responsible for)那个 key 的存储节点。

4.2 划分算法

Dynamo 的关键设计要求之一是必须增量可扩展性。这就需要一个机制来将数据动态划分到系统中的节点(即存储主机)上去。Dynamo 的分区方案依赖于一致哈希将负载分发到多个存储主机。在一致的哈希中[10]，一个哈希函数的输出范围被视为一个固定的圆形空间或“环”(即最大的哈希值绕到(wrap)最小的哈希值)。系统中的每个节点被分配了这个空间中的一个随机值，它代表着它的在环上的“位置”。每个由 key 标识的数据项通过计算数据项的 key 的 hash 值来产生其在环上的位置。然后沿顺时针方向找到第一个其位置比计算的数据项的位置大的节点。因此，每个节点变成了环上的一个负责它自己与它的前身节点间的**区域**(region)。一致性哈希的主要优点是节点的进进出出(departure or arrival)只影响其最直接的邻居，而对其他节点没影响。

这对基本的一致性哈希算法提出了一些挑战。首先，每个环上的任意位置的节点分配导致非均匀的数据和负荷分布。二，基本算法无视于节点的性能的异质性。为了解决这些问题，Dynamo 采用了一致的哈希(类似于[10，20]中使用的)的变体：每个节点被分配到环多点而不是映射到环上的一个单点。为此，Dynamo 使用了“虚拟节点”的概念。系统中一个虚拟节点看起来像单个节点，但每个节点可对多个虚拟节点负责。实际上，当一个新的节点添加到系统中，它被分配环上的多个位置(以下简称“标记” Token)。对 Dynamo 的划分方案进一步细化在第 6 部分讨论。

使用虚拟节点具有以下优点：

如果一个节点不可用(由于故障或日常维护)，这个节点处理的负载将均匀地分散在剩余的可用节点。当一个节点再次可用，或一个新的节点添加到系统中，新的可用节点接受来自其他可用的每个节点的负载量大致相当。一个节点负责的虚拟节点的数目可以根据其处理能力来决定，顾及到物理基础设施的异质性。

4.3 复制

为了实现高可用性和耐用性，Dynamo 将数据复制到多台主机上。每个数据项被复制到 N 台主机，其中 N 是“*每实例*”(“*per-instance)的*配置参数。每个键，*K*，被分配到一个协调器(coordinator)节点(在上一节所述)。协调器节点掌控其负责范围内的复制数据项。**除了在本地存储其范围内的每个 key 外，协调器节点复制这些 key 到环上顺时针方向的 N-1 后继节点。这样的结果是，系统中每个节点负责环上的从其自己到第 N 个前继节点间的一段区域**。在图 2 中，节点 B 除了在本地存储键 *K 外*,在节点 C 和 D 处复制键 *K*。节点 D 将存储落在范围(A,B],(B,C]和(C,D]上的所有键。

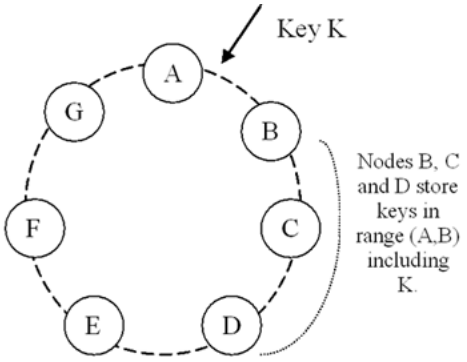

图 2：Dynamo 的划分和键的复制。

一个负责存储一个特定的键的节点列表被称为*首选列表(preference list)*。该系统的设计，如将 4.8 节中解释，让系统中每一个节点可以决定对于任意 key 哪些节点应该在这个清单中。出于对节点故障的考虑，首选清单可以包含起过 N 个节点。请注意，因使用虚拟节点，对于一个特定的 key 的第一个 N 个后继位置可能属于少于 N 个物理所节点(即节点可以持有多个第一个 N 个位置)。为了解决这个问题，一个 key *首选列表*的构建将跳过环上的一些位置，以确保该列表只包含不同的物理节点。

4.4 版本的数据

Dynamo 提供最终一致性，从而允许更新操作可以异步地传播到所有副本。put()调用可能在更新操作被所有的副本执行之前就返回给调用者，这可能会导致一个场景：在随后的 get()操作可能会返回一个不是最新的对象。如果没有失败，那么更新操作的传播时间将有一个上限。但是，在某些故障情况下(如服务器故障或网络 partitions)，更新操作可能在一个较长时间内无法到达所有的副本。

在 Amazon 的平台，有一种类型的应用可以容忍这种不一致，并且可以建造并操作在这种条件下。例如，购物车应用程序要求一个“*添加到购物车*“动作从来没有被忘记或拒绝。如果购物车的最近的状态是不可用，并且用户对一个较旧版本的购物车做了更改，这种变化仍然是有意义的并且应该保留。但同时它不应取代当前不可用的状态，而这不可用的状态本身可能含有的变化也需要保留。请注意在 Dynamo 中“*添加到购物车*“和”从*购物车删除项目*“这两个操作被转成 put 请求。当客户希望增加一个项目到购物车(或从购物车删除)但最新的版本不可用时，该项目将被添加到旧版本(或从旧版本中删除)并且不同版本将在后来协调(reconciled)。

为了提供这种保证，Dynamo 将每次数据修改的结果当作一个新的且不可改变的数据版本。它允许系统中同一时间出现多个版本的对象。大多数情况，新版本包括（subsume）老的版本，且系统自己可以决定权威版本(语法协调 syntactic reconciliation)。然而，版本分支可能发生在并发的更新操作与失败的同时出现的情况，由此产生冲突版本的对象。在这种情况下，系统无法协调同一对象的多个版本，那么客户端必须执行协调，将多个分支演化后的数据崩塌(collapse)成一个合并的版本(语义协调)。一个典型的崩塌的例子是“合并”客户的不同版本的购物车。使用这种协调机制，一个“添加到购物车”操作是永远不会丢失。但是，**已删除的条目可能会”重新浮出水面”（resurface）**。

重要的是要了解某些故障模式有可能导致系统中相同的数据不止两个，而是好几个版本。在网络分裂和节点故障的情况下，可能会导致一个对象有不同的分历史，系统将需要在未来协调对象。这就要求我们在设计应用程序，明确意识到相同数据的多个版本的可能性(以便从来不会失去任何更新操作)。

Dynamo 使用矢量时钟[12]来捕捉同一不同版本的对象的因果关系。矢量时钟实际上是一个(node,counter)对列表(即(节点，计数器)列表)。矢量时钟是与每个对象的每个版本相关联。**通过审查其向量时钟，我们可以判断一个对象的两个版本是平行分枝或有因果顺序**。如果第一个时钟对象上的计数器在第二个时钟对象上小于或等于其他所有节点的计数器，那么第一个是第二个的祖先，可以被人忽略。否则，这两个变化被认为是冲突，并要求协调。

在 dynamo 中，当客户端更新一个对象，它必须指定它正要更新哪个版本。这是通过传递它从早期的读操作中获得的上下文对象来指定的，它包含了向量时钟信息。当处理一个读请求，如果 Dynamo 访问到多个不能语法协调(syntactically reconciled)的分支，它将返回分支叶子处的所有对象，其包含与上下文相应的版本信息。使用这种上下文的更新操作被认为已经协调了更新操作的不同版本并且分支都被倒塌到一个新的版本。

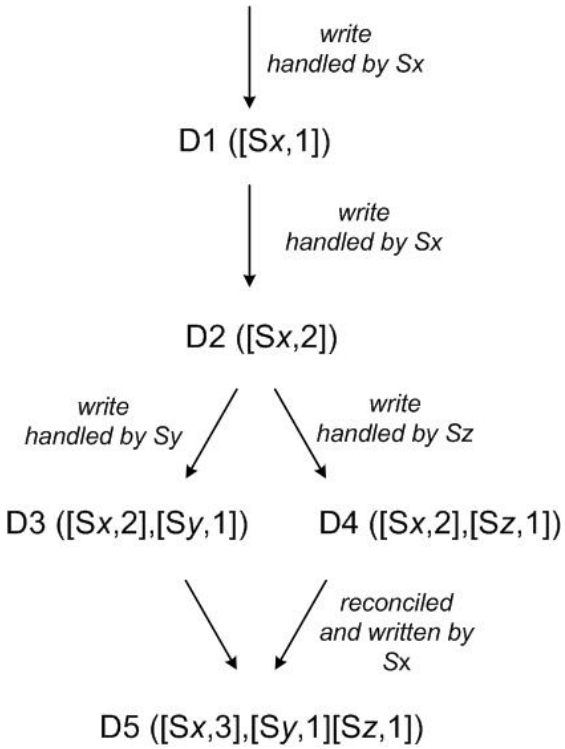

图 3：对象的版本随时间演变。

为了说明使用矢量时钟，让我们考虑图 3 所示的例子。

1)客户端写入一个新的对象。节点(比如说 Sx)，它处理对这个 key 的写：序列号递增，并用它来创建数据的向量时钟。该系统现在有对象 D1 和其相关的时钟[(Sx，1)]。

2)客户端更新该对象。假定也由同样的节点处理这个要求。现在该系统有对象 D2 和其相关的时钟[(Sx，2)]。D2 继承自 D1，因此覆写 D1，但是节点中或许存在还没有看到 D2 的 D1 的副本。

3)让我们假设，同样的客户端更新这个对象但不同的服务器(比如 Sy)处理了该请求。目前该系统具有数据 D3 及其相关的时钟[(Sx，2)，(Sy，1)]。

4)接下来假设不同的客户端读取 D2，然后尝试更新它，并且另一个服务器节点(如 Sz)进行写操作。该系统现在具有 D4(D2 的子孙)，其版本时钟[(Sx，2)，(Sz，1)]。一个对 D1 或 D2 有所了解的节点可以决定，在收到 D4 和它的时钟时，新的数据将覆盖 D1 和 D2，可以被垃圾收集。一个对 D3 有所了

解的节点，在接收 D4 时将会发现，它们之间不存在因果关系。换句话说，**D3 和 D4 都有更新操作，但都未在对方的变化中反映出来**。这两个版本的数据都必须保持并提交给客户端(在读时)进行语义协调。

5)现在假定一些客户端同时读取到 D3 和 D4(上下文将反映这两个值是由 read 操作发现的)。读的上下文包含有 D3 和 D4 时钟的概要信息，即[(Sx，2)，(Sy，1)，(Sz，1)]的时钟总结。如果客户端执行协调，且由节点 Sx 来协调这个写操作，Sx 将更新其时钟的序列号。D5 的新数据将有以下时钟:[(Sx，3)，(Sy，1)，(Sz，1)]。

关于向量时钟一个可能的问题是，如果许多服务器协调对一个对象的写，向量时钟的大小可能会增长。实际上，这是不太可能的，因为写入通常是由首选列表中的前 N 个节点中的一个节点处理。在网络分裂或多个服务器故障时，写请求可能会被不是首选列表中的前 N 个节点中的一个处理的，因此会导致矢量时钟的大小增长。在这种情况下，值得限制向量时钟的大小。为此，Dynamo 采用了以下时钟截断方案：伴随着每个(节点，计数器)对，Dynamo 存储一个时间戳表示最后一次更新的时间。当向量时钟中(节点，计数器)对的数目达到一个阈值(如 10)，最早的一对将从时钟中删除。显然，这个截断方案会导至在协调时效率低下，因为后代关系不能准确得到。不过，这个问题还没有出现在生产环境，因此这个问题没有得到彻底研究。

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的任何对 key 的 get 和 put 操作。在本节中，对简单起见，我们将描述如何在一个从不失败的(failure-free)环境中执行这些操作，并在随后的章节中，我们描述了在故障的情况下读取和写入操作是如何执行。

GET 和 PUT 操作都使用基于 Amazon 基础设施的特定要求，通过 HTTP 的处理框架来调用。一个客户端可以用有两种策略之一来选择一个节点：(1)通过一个普通的负载平衡器路由请求，它将根据负载信息选择一个节点，或(2)使用一个分区(partition)敏感的客户端库直接路由请求到适当的协调程序节点。第一个方法的优点是，客户端没有链接(link)任何 Dynamo 特定的代码在到其应用中，而第二个策略，Dynamo 可以实现较低的延时，因为它跳过一个潜在的转发步骤。

处理读或写操作的节点被称为***协调员***。通常，这是首选列表中跻身前 N 个节点中的第一个。如果请求是通过负载平衡器收到，访问 key 的请求可能被路由到环上任何随机节点。在这种情况下，如果接收到请求节点不是请求的 key 的首选列表中前 N 个节点之一，它不会协调处理请求。相反，该节点将请求转发到首选列表中第一个跻身前 N 个节点。

读取和写入操作涉及到首选清单中的前 N 个健康节点，跳过那些瘫痪的(down)或者不可达(inaccessible)的节点。当所有节点都健康，key 的首选清单中的前 N 个节点都将被访问。当有节点故障或网络分裂，首选列表中排名较低的节点将被访问。

为了保持副本的一致性，Dynamo 使用的一致性协议类似于仲裁(quorum)。该协议有两个关键配置值：R 和 W. **R 是必须参与一个成功的读取操作的最少数节点数目。W 是必须参加一个成功的写操作的最少节点数**。设定 R 和 W，使得 **R+W>N** 产生类似仲裁的系统。在此模型中，一个 get(or out)操作延时是由最慢的 R(或 W)副本决定的。基于这个原因，R 和 W 通常配置为小于 N，为客户提供更好的延时。

当收到对 key 的 put()请求时，协调员生成新版本向量时钟并在本地写入新版本。协调员然后将新版本(与新的向量时钟一起)发送给首选列表中的排名前 N 个的可达节点。如果至少 W-1 个节点返回了响应，那么这个写操作被认为是成功的。

同样，对于一个 get()请求，协调员为 key 从首选列表中排名前 N 个可达节点处请求所有现有版本的数据，然后等待 R 个响应，然后返回结果给客户端。如果最终协调员收集的数据的多个版本，它返回所有它认为没有因果关系的版本。不同版本将被协调，并且取代当前的版本，最后写回。

4.6 故障处理：暗示移交(Hinted Handoff)

Dynamo 如果使用传统的仲裁(quorum)方式,在服务器故障和网络分裂的情况下它将是不可用，即使在最简单的失效条件下也将降低耐久性。为了弥补这一点，它不严格执行仲裁，即使用了“马虎仲裁”(“sloppy quorum”)，所有的读，写操作是由首选列表上的前 N 个*健康的*节点执行的，它们可能不总是在散列环上遇到的那前 N 个节点。

考虑在图 2 例子中 Dynamo 的配置，给定 N=3。在这个例子中，如果写操作过程中节点 A 暂时 Down 或无法连接，然后通常本来在 A 上的一个副本现在将发送到节点 D。这样做是为了保持期待的可用性和耐用性。发送到 D 的副本在其原数据中将有一个**暗示，表明哪个节点才是在副本预期的接收者**(在这种情况下 A)。接收暗示副本的节点将数据保存在一个单独的本地存储中，他们被定期扫描。在检测到了 A 已经复苏，D 会尝试发送副本到 A。一旦传送成功，D 可将数据从本地存储中删除而不会降低系统中的副本总数。

使用暗示移交，Dynamo 确保读取和写入操作不会因为节点临时或网络故障而失败。需要最高级别的可用性的应用程序可以设置 W 为 1，这确保了只要系统中有一个节点将 key 已经持久化到本地存储 ，一个写是可以接受(**即一个写操作完成即意味着成功**)。因此，只有系统中的所有节点都无法使用时写操作才会被拒绝。然而，在实践中，大多数 Amazon 生产服务设置了更高的 W 来满足耐久性极别的要求。对 N, R 和 W 的更详细的配置讨论在后续的第 6 节。

一个高度可用的存储系统具备处理整个数据中心故障的能力是非常重要的。数据中心由于断电，冷却装置故障，网络故障和自然灾害发生故障。Dynamo 可以配置成跨多个数据中心地对每个对象进行复制。从本质上讲，一个 key 的首选列表的构造是基于跨多个数据中心的节点的。这些数据中心通过高速网络连接。这种跨多个数据中心的复制方案使我们能够处理整个数据中心故障。

4.7 处理永久性故障：副本同步

Hinted Handoff 在系统成员流动性(churn)低，节点短暂的失效的情况下工作良好。有些情况下，在 hinted 副本移交回原来的副本节点之前，暗示副本是不可用的。为了处理这样的以及其他威胁的耐久性问题，Dynamo 实现了反熵(anti-entropy，或叫副本同步)协议来保持副本同步。

为了更快地检测副本之间的不一致性，并且减少传输的数据量，Dynamo 采用 MerkleTree[13]。MerkleTree 是一个哈希树(Hash Tree)，其叶子是各个 key 的哈希值。树中较高的父节点均为其各自孩子节点的哈希。该 merkleTree 的主要优点是树的每个分支可以独立地检查，而不需要下载整个树或整个数据集。此外，MerkleTree 有助于减少为检查副本间不一致而传输的数据的大小。例如，如果两树的根哈希值相等，且树的叶节点值也相等，那么节点不需要同步。如果不相等，它意味着，一些副本的值是不同的。在这种情况下，节点可以交换 children 的哈希值，处理直到它到达了树的叶

子，此时主机可以识别出“不同步”的 key。MerkleTree 减少为同步而需要转移的数据量，减少在反熵过程中磁盘执行读取的次数。

Dynamo 在反熵中这样使用 MerkleTree：每个节点为它承载的每个 key 范围(由一个虚拟节点覆盖 key 集合)维护一个单独的 MerkleTree。这使得节点可以比较 key range 中的 key 是否是最新。在这个方案中，两个节点交换 MerkleTree 的根，对应于它们承载的共同的键范围。其后，使用上面所述树遍历方法，节点确定他们是否有任何差异和执行适当的同步行动。方案的缺点是，当节点加入或离开系统时有许多 key rangee 变化，从而需要重新对树进行计算。通过由 6.2 节所述的更精炼 partitioning 方案，这个问题得到解决。

4.8 会员和故障检测 4.8.1 环会员(Ring Membership)

Amazon 环境中，节点中断(由于故障和维护任务)常常是暂时的，但持续的时间间隔可能会延长。一个节点故障很少意味着一个节点永久离开，因此应该不会导致对已分配的分区重新平衡(rebalancing)和修复无法访问的副本。同样，人工错误可能导致意外启动新的 Dynamo 节点。基于这些原因，应当适当使用一个明确的机制来发起节点的增加和从环中移除节点。管理员使用命令行工具或浏览器连接到一个节点，并发出成员改变(membership change)指令指示一个节点加入到一个环或从环中删除一个节点。接收这一请求的节点写入成员变化以及适时写入持久性存储。该成员的变化形成了历史，因为节点可以被删除，重新添加多次。一个基于 Gossip 的协议传播成员变动，并维持成员的最终一致性。每个节点每间隔一秒随机选择随机的对等节点，两个节点有效地协调他们持久化的成员变动历史。

当一个节点第一次启动时，它选择它的 Token(在虚拟空间的一致哈希节点) 并将节点映射到各自的 Token 集(Token set)。该映射被持久到磁盘上，最初只包含本地节点和 Token 集。在不同的节点中存储的映射（节点到 token set 的映射）将在协调成员的变化历史的通信过程中一同被协调。因此，划分和布局信息也是基于 Gossip 协议传播的，因此每个存储节点都了解对等节点所处理的标记范围。这使得每个节点可以直接转发一个 key 的读/写操作到正确的数据集节点。

4.8.2 外部发现

上述机制可能会暂时导致**逻辑分裂**的 Dynamo 环。例如，管理员可以将节点 A 加入到环，然后将节点 B 加入环。在这种情况下，节点 A 和 B 各自都将认为自己是环的一员，但都不会立即了解到其他的节点(**也就是 A 不知道 B 的存在，B 也不知道 A 的存在，这叫逻辑分裂**)。为了防止逻辑分裂，有些 Dynamo 节点扮演**种子节点**的角色。种子的发现(discovered)是通过外部机制来实现的并且所有其他节点都知道(实现中可能直接在配置文件中指定 seed node 的 IP，或者实现一个动态配置服务,seed register)。因为所有的节点，最终都会和种子节点协调成员关系，逻辑分裂是极不可能的。种子可从静态配置或配置服务获得。通常情况下，种子在 Dynamo 环中是一个全功能节点。

4.8.3 故障检测

Dynamo 中，故障检测是用来避免在进行 get()和 put()操作时尝试联系无法访问节点，同样还用于分区转移(transferring partition)和暗示副本的移交。为了避免在通信失败的尝试，一个纯本地概念的失效检测完全足够了：如果节点 B 不对节点 A 的信息进行响应(即使 B 响应节点 C 的消息)，节点 A 可能会认为节点 B 失败。在一个客户端请求速率相对稳定并产生节点间通信的 Dynamo 环中，一个节点 A 可以快速发现另一个节点 B 不响应时，节点 A 则使用映射到 B 的分区的备用节点服务请求，并定期检查节点 B 后来是否后来被复苏。在没有客户端请求推动两个节点之间流量的情况下，节点双方并不真正需要知道对方是否可以访问或可以响应。

去中心化的故障检测协议使用一个简单的 Gossip 式的协议，使系统中的每个节点可以了解其他节点到达(或离开)。有关去中心化的故障探测器和影响其准确性的参数的详细信息，感兴趣的读者可以参考[8]。早期 Dynamo 的设计使用去中心化的故障检测器以维持一个失败状态的全局性的视图。后来认为，显式的节点加入和离开的方法排除了对一个失败状态的全局性视图的需要。这是因为节点是是可以通过节点的显式加入和离开的方法知道节点永久性(permanent)增加和删除，而短暂的(temporary)节点失效是由独立的节点在他们不能与其他节点通信时发现的(当转发请求时)。

4.9 添加/删除存储节点

当一个新的节点(例如 X)添加到系统中时，它被分配一些随机散落在环上的 Token。对于每一个分配给节点 X 的 key range，当前负责处理落在其 key range 中的 key 的节点数可能有好几个(小于或等于 N)。由于 key range 的分配指向 X，一些现有的节点不再需要存储他们的一部分 key，这些节点将这些 key 传给 X，让我们考虑一个简单的引导(bootstrapping)场景，节点 X 被添加到图 2 所示的环中 A 和 B 之间，当 X 添加到系统，它负责的 key 范围为(F,G]，(G，A]和(A，X]。因此，节点 B,C 和 D 都各自有一部分不再需要储存 key 范围(在 X 加入前，B 负责(F,G]，(G,A]，(A,B]；C 负责(G,A]，(A,B]，(B,C]；D 负责(A,B]，(B,C]，(C,D]。而在 X 加入后，B 负责(G,A]，(A,X]，(X,B]；C 负责(A,X]，(X,B]，(B,C]；D 负责(X,B]，(B,C]，(C,D])。因此，节点 B，C 和 D，当收到从 X 来的确认信号时将供出(offer)适当的 key。当一个节点从系统中删除，key 的重新分配情况按一个相反的过程进行。

实际经验表明，这种方法可以将负载均匀地分布到存储节点，其重要的是满足了延时要求，且可以确保快速引导。最后，在源和目标间增加一轮确认(confirmation round)以确保目标节点不会重复收到任何一个给定的 key range 转移。

5 实现

在 dynamo 中，每个存储节点有三个主要的软件组件：请求协调，成员(membership)和故障检测，以及本地持久化引擎。所有这些组件都由 Java 实现。

Dynamo 的本地持久化组件允许插入不同的存储引擎，如：Berkeley 数据库(BDB 版本)交易数据存储，BDB Java 版，MySQL，以及一个具有持久化后备存储的内存缓冲。设计一个可插拔的持久化组件的主要理由是要按照应用程序的访问模式选择最适合的存储引擎。例如，BDB 可以处理的对象通常为几十千字节的数量级，而 MySQL 能够处理更大尺寸的对象。应用根据其对象的大小分布选择相应的本地持久性引擎。生产中，Dynamo 多数使用 BDB 事务处理数据存储。

请求协调组成部分是建立在事件驱动通讯基础上的，其中消息处理管道分为多个阶段类似 SEDA 的结构[24]。所有的通信都使用 Java NIO Channels。协调员执行读取和写入：通过收集从一个或多个节点数据(在读的情况下)，或在一个或多个节点存储的数据(写入)。每个客户的请求中都将导致在收到客户端请求的节点上一个状态机的创建。**每一个状态机包含以下逻辑**：标识负责一个 key 的节点，发送请求，等待回应，可能的重试处理，加工和包装返回客户端响应。每个状态机实例只处理一个客户端请求。例如，一个**读操作**实现了以下状态机：(i)发送读请求到相应节点，(ii)等待所需的最低数量的响应，(iii)如果在给定的时间内收到的响应太少，那么请求失败，(iv)否则，收集所有数据的版本，并确定要返回的版本 (v)如果启用了版本控制，执行语法协调，并产生一个对客户端不透明写上下文，其包括一个涵括所有剩余的版本的矢量时钟。为了简洁起见，没有包含故障处理和重试逻辑。

在读取响应返回给调用方后，状态机等待一小段时间以接受任何悬而未决的响应。如果任何响应返回了过时了的(stale)的版本，协调员将用最新的版本更新这些节点(当然是在后台了)。这个过程被称为*读修复(read repair)*，因为它是用来修复一个在某个时间曾经错过更新操作的副本，同时 read repair 可以消除不必的反熵操作。

如前所述，**写请求**是由首选列表中某个排名前 N 的节点来协调的。虽然总是选择前 N 节点中的第一个节点来协调是可以的，但在单一地点序列化所有的写的做法会导致负荷分配不均，进而导致违反 SLA。为了解决这个问题，首选列表中的前 N 的任何节点都允许协调。特别是，由于写通常跟随在一个读操作之后，写操作的协调员将由节点上最快答复之前那个读操作的节点来担任，这是因为这些信息存储在请求的上下文中(指的是 write 操作的请求)。这种优化使我们能够选择那个存有同样被之前读操作使用过的数据的节点，从而提高“读你的写”(read-your-writes)一致性(译：我不认为这个描述是有道理的，因为作者这里描述明明是 write-follows-read,要了解 read-your-writes 一致性的读者参见作者另一篇文章:eventually consistent)。它也减少了为了将处理请求的性能提高到 99.9 百分位时性能表现的差异。

6 经验与教训

Dynamo 由几个不同的配置的服务使用。这些实例有着不同的版本协调逻辑和读/写仲裁(quorum)的特性。以下是 Dynamo 的主要使用模式：

*业务逻辑特定的协调：*这是一个普遍使用的 Dynamo 案例。每个数据对象被复制到多个节点。在版本发生分岔时，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是这一类的典型例子。其业务逻辑是通过合并不同版本的客户的购物车来协调不同的对象。

*基于时间戳的协调：*此案例不同于前一个在于协调机制。在出现不同版本的情况下，Dynamo 执行简单的基于时间戳的协调逻辑：“最后的写获胜”，也就是说，具有最大时间戳的对象被选为正确的版本。一些维护客户的会话信息的服务是使用这种模式的很好的例子。

*高性能读取引擎：*虽然 Dynamo 被构建成一个“永远可写”数据存储，一些服务通过调整其仲裁的特性把它作为一个高性能读取引擎来使用。通常，这些服务有很高的读取请求速率但只有少量的更新操作。在此配置中，通常 R 是设置为 1，且 W 为 N。对于这些服务，Dynamo 提供了划分和跨多个节点的复制能力，从而提供增量可扩展性(incremental scalability)。一些这样的实例被当成权威数据缓存用来缓存重量级后台存储的数据。那些保持产品目录及促销项目的服务适合此种类别。

Dynamo 的主要优点是它的客户端应用程序可以调的 N，R 和 W 的值，以实现其期待的性能，可用性和耐用性的水平。例如，**N 的值决定了每个对象的耐久性。Dynamo 用户使用的一个典型的 N 值是 3**。

W 和 R 影响对象的可用性，耐用性和一致性。举例来说，如果 W 设置为 1，只要系统中至少有一个节点活就可以成功地处理一个写请求，那么系统将永远不会拒绝写请求。不过，低的 W 和 R 值会增加不一致性的风险，因为写请求被视为成功并返回到客户端，即使他们还未被大多数副本处理。这也引入了一个耐用性漏洞(vulnerability)窗口：即使它只是在少数几个节点上持久化了但写入请求成功返回到客户端。

传统的观点认为，耐用性和可用性关系总是非常紧密(hand-in-hand 手牵手^-^)。但是，这并不一定总是真的。例如，耐用性漏洞窗口可以通过增加 W 来减少，但这将增加请求被拒绝的机率(从而减少可用性)，因为为处理一个写请求需要更多的存储主机需要活着。

被好几个 Dynamo 实例采用的(n，R，W)配置通常为(3,2,2)。选择这些值是为满足性能，耐用性，一致性和可用性 SLAs 的需求。

所有在本节中测量的是一个在线系统，其工作在(3,2,2)配置并运行在几百个同质硬件配置上。如前所述，每一个实例包含位于多个数据中心的 Dynamo 节点。这些数据中心通常是通过高速网络连接。回想一下，产生一个成功的 get(或 put)响应，R(或 W)个节点需要响应协调员。显然，数据中心之间的网络延时会影响响应时间，因此节点(及其数据中心位置)的选择要使得应用的目标 SLAs 得到满足。

6.1 平衡性能和耐久性

虽然 Dynamo 的主要的设计目标是建立一个高度可用的数据存储，性能是在 Amazon 平台中是一个同样重要的衡量标准。如前所述，为客户提供一致的客户体验，Amazon 的服务定在较高的百分位(如 99.9 或 99.99)，一个典型的使用 Dynamo 的服务的 SLA 要求 99.9%的读取和写入请求在 300 毫秒内完成。

由于 Dynamo 是运行在标准的日用级硬件组件上，这些组件的 I/O 吞吐量远比不上高端企业级服务器，因此提供一致性的高性能的读取和写入操作并不是一个简单的任务。再加上涉及到多个存储节点的读取和写入操作，让我们更加具有挑战性，因为这些操作的性能是由最慢的 R 或 W 副本限制的。图 4 显示了 Dynamo 为期 30 天的读/写的平均和 99.9 百分位的延时。正如图中可以看出，延时表现出明显的昼夜模式这是因为进来的请求速率存在昼夜模式的结果造成的(即请求速率在白天和黑夜有着显着差异)。此外，写延时明显高于读取延时，因为写操作总是导致磁盘访问。此外，99.9 百分位的延时大约是 200 毫秒，比平均水平高出一个数量级。这是因为 99.9 百分位的延时受几个因素，如请求负载，对象大小和位置格局的变化影响。

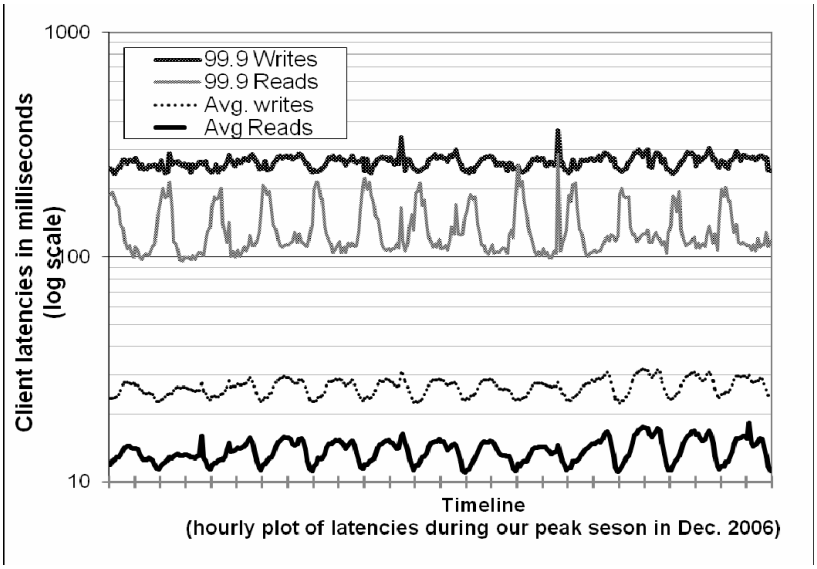

图 4：读，写操作的平均和 99.9 百分点延时，2006 年 12 月高峰时的请求。

在 X 轴的刻度之间的间隔相当于连续 12 小时。延时遵循昼夜模式类似请求速率 99.9 百分点比平均水平高出一个数量级。

虽然这种性能水平是可以被大多数服务所接受，一些面向客户的服务需要更高的性能。针对这些服务，Dynamo 能够牺牲持久性来保证性能。在这个优化中，每个存储节点维护一个内存中的对象缓冲区（BigTable 中的 memtable）。每次写操作都存储在缓冲区，“写”线程定期将缓冲写到存储中。在这个方案中，读操作首先检查请求的 key 是否存在于缓冲区。如果是这样，对象是从缓冲区读取，而不是存储引擎。

这种优化的结果是 99.9 百位在流量高峰期间的延时降低达 5 倍之多，即使是一千个对象(参见图 5)的非常小的缓冲区。此外，如图中所示，写缓冲在较高百分位具有平滑延时。显然，这个方案是平衡耐久性来提高性能的。在这个方案中，**服务器崩溃可能会导致写操作丢失，即那些在缓冲区队列中的写(还未持久化到存储中的写)**。为了减少耐用性风险，更细化的写操作要求协调员选择 N 副本中的一个执行“**持久写**”。由于协调员只需等待 W 个响应(译，这里讨论的这种情况包含 W-1 个缓冲区写，1 个持久化写)，写操作的性能不会因为单一一个副本的持久化写而受到影响。

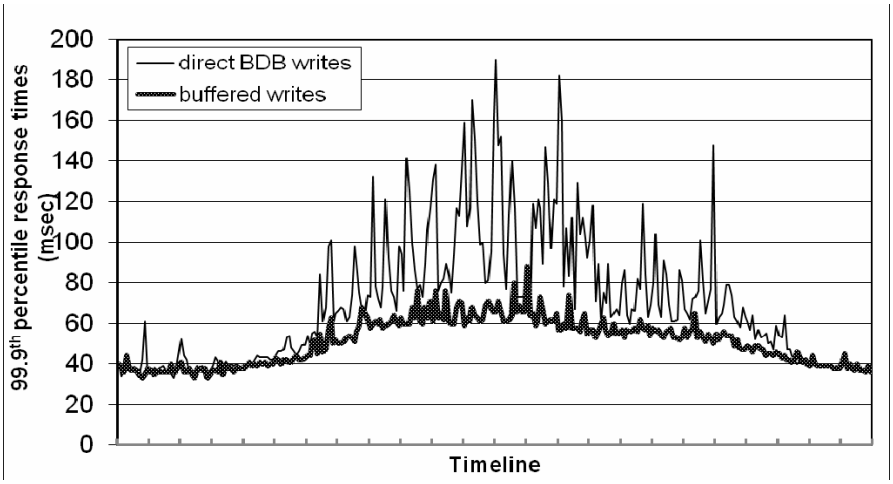

图 5：24 小时内的 99.9 百分位延时缓冲和非缓冲写的性能比较。在 x 轴的刻度之间的间隔连续为一小时。

6.2 确保均匀的负载分布

Dynamo 采用一致性的散列将 key space(键空间)分布在其所有的副本上，并确保负载均匀分布。假设对 key 的访问分布不会高度偏移，一个统一的 key 分配可以帮助我们达到均匀的负载分布。特别地，Dynamo 设计假定，即使访问的分布存在显着偏移，只要在流行的那端(popular end)有足够多的 keys，那么对那些流行的 key 的处理的负载就可以通过 partitioning 均匀地分散到各个节点。本节讨论 Dynamo 中所出现负载不均衡和不同的划分策略对负载分布的影响。

为了研究负载不平衡与请求负载的相关性，通过测量各个节点在 24 小时内收到的请求总数-细分为 30 分钟一段。在一个给定的时间窗口，如果该节点的请求负载偏离平均负载没有超过某个阈值(这里 15%)，认为一个节点被认为是“平衡的”。否则，节点被认为是“失去平衡”。图 6 给出了一部分在这段时间内“失去平衡”的节点(以下简称“失衡比例”)。作为参考，整个系统在这段时间内收到的相应的请求负载也被绘制。正如图所示，不平衡率随着负载的增加而下降。例如，在低负荷时，不平衡率高达 20%，在高负荷高接近 10%。直观地说，这可以解释为，在高负荷时大量流行键(popular key)访问且由于 key 的均匀分布，负载最终均匀分布。然而，在(其中负载为高峰负载的八分之一)低负载下，当更少的流行键被访问，将导致一个比较高的负载不平衡。

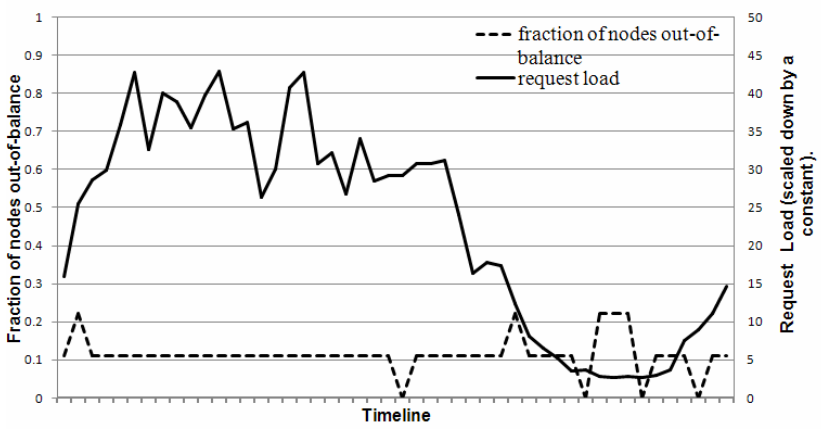

图 6：部分失去平衡的节点(即节点的请求负载高于系统平均负载的某一阈值)和其相应的请求负载。

X 轴刻度间隔相当于一个 30 分钟的时间。

本节讨论 Dynamo 的划分方案(partitioning scheme)是如何随着时间和负载分布的影响进行演化的。

策略 1：每个节点 T 个随机 Token 和基于 Token 值进行分割：这是最早部署在生产环境的策略(在 4.2 节中描述)。在这个方案中，每个节点被分配 T 个 Tokens(从哈希空间随机均匀地选择)。所有节点的 token，是按照其在哈希空间中的值进行排序的。每两个连续的 Token 定义一个范围。最后的 Token 与最开始的 Token 构成一区域(range)：从哈希空间中最大值绕(wrap)到最低值。由于 Token 是随机选择，范围大小是可变的。节点加入和离开系统导致 Token 集的改变，最终导致 ranges 的变化，请注意，每个节点所需的用来维护系统的成员的空间与系统中节点的数目成线性关系。

在使用这一策略时，遇到了以下问题。首先，当一个新的节点加入系统时，它需要“窃取”(steal)其他节点的键范围。然而，这些需要移交 key ranges 给新节点的节点必须扫描他们的本地持久化存储来得到适当的数据项。请注意，在生产节点上执行这样的扫描操作是非常复杂，因为扫描是资源高度密集的操作，他们需要在后台执行，而不至于影响客户的性能。这就要求我们必须将引导工作设置为最低的优先级。然而，这将大大减缓了引导过程，在繁忙的购物季节，当节点每天处理数百万的请求时，引导过程可能需要几乎一天才能完成。第二，当一个节点加入/离开系统，由许多节点处理的 key range 的变化以及新的范围的 MertkleTree 需要重新计算，在生产系统上，这不是一个简单的操作。最后，由于 key range 的随机性，没有一个简单的办法为整个 key space 做一个快照，这使得归档过程复杂化。在这个方案中，归档整个 key space 需要分别检索每个节点的 key，这是非常低效的。

这个策略的根本问题是，数据划分和数据安置的计划交织在一起。例如，在某些情况下，最好是添加更多的节点到系统，以应对处理请求负载的增加。但是，在这种情况下，添加节点(导致数据安置)不可能不影响数据划分。理想的情况下，最好使用独立划分和安置计划。为此，对以下策略进行了评估：

*策略 2：每个节点 T 个随机 token 和同等大小的分区：*在此策略中，节点的哈希空间分为 Q 个同样大小的分区/范围，每个节点被分配 T 个随机 Token。Q 是通常设置使得 Q>>N 和 Q>>S*T，其中 S 为系统的节点个数。在这一策略中，**Token 只是用来构造一个映射函数**该函数将哈希空间的值映射到一个有序列的节点列表，**而不决定分区**。分区是放置在从分区的末尾开始沿着一致性 hash 环顺时针移动遇到的前 N 个独立的节点上。图 7 说明了这一策略当 N=3 时的情况。在这个例子中，节点 A，B，C 是

从分区的末尾开始沿着一致性 hash 环顺时针移动遇到的包含 key K1 的节点。这一策略的主要优点是：(i)划分和分区布局脱耦 (ii)使得在运行时改变安置方案成为可能。

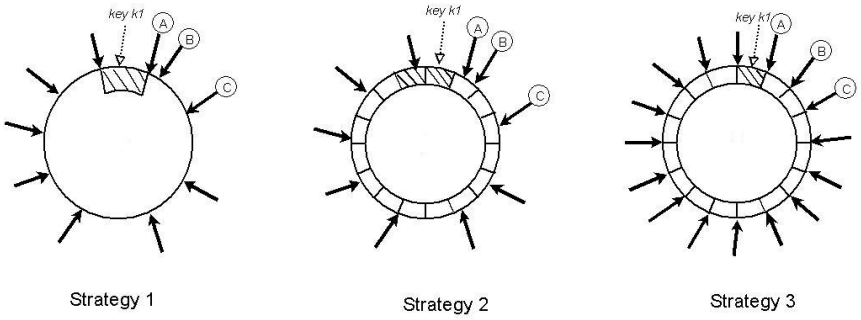

图 7：三个策略的分区和 key 的位置。
甲，乙，丙描述三个独立的节点，形成 keyk1 在一致性哈希环上的首选列表(N＝3)。
阴影部分表示节点 A，B 和 C 形式的首选列表负责的 keyrangee。
黑色箭头标明各节点的 Token 的位置。

*策略 3：每个节点 Q/S 个 Token，大小相等的分区：*类似策略 2，这一策略空间划分成同样大小为 Q 的散列分区，以及分区布局(placement of partition)与划分方法(partitioning scheme)脱钩。此外，每个节点被分配 Q/S 个 Token 其中 **S 是系统的节点数**。当一个节点离开系统，为使这些属性被保留，它的 Token 随机分发到其他节点。同样，当一个节点加入系统，新节点将通过一种可以保留这种属性的方式从系统的其他节点“偷”Token。

对这三个策略的效率评估使用 S=30 和 N=3 配置的系统。然而，以一个比较公平的方式这些不同的策略是很难的，因为不同的策略有不同的配置来调整他们的效率。例如，策略 1 取决于负荷的适当分配(即 T)，而策略 3 信赖于分区的个数(即 Q)。一个公平的比较方式是在所有策略中使用相同数量的空间来维持他们的成员信息时，通过评估负荷分布的偏斜. 例如，策略 1 每个节点需要维护所有环内的 Token 位置，策略 3 每个节点需要维护分配到每个节点的分区信息。

在我们的下一个实验，通过改变相关的参数(T 和 Q),对这些策略进行了评价。每个策略的负载均衡的效率是根据每个节点需要维持的成员信息的大小的不同来测量，*负载平衡效率*是指每个节点服务的平均请求数与最忙(hottest)的节点服务的最大请求数之比。

结果示于图 8。正如图中看到，**策略 3 达到最佳的负载平衡效率**，而策略 2 最差负载均衡的效率。一个短暂的时期，在将 Dynamo 实例从策略 1 到策略 3 的迁移过程中，策略 2 曾作为一个临时配置。相对于策略 1,策略 3 达到更好的效率并且在每个节点需要维持的信息的大小规模降低了三个数量级。虽然存储不是一个主要问题，但节点间周期地 Gossip 成员信息，因此最好是尽可能保持这些信息紧凑。除了这个，**策略 3 有利于且易于部署，理由如下**：(i)*更快的 bootstrapping/恢复*：由于分区范围是固定的，它们可以被保存在单独的文件，这意味着一个分区可以通过简单地转移文件并作为一个单位重新安置(避免随机访问需要定位具体项目)。这简化了引导和恢复过程。(ii)*易于档案*：对数据集定期归档是 Amazon 存储服务提出的强制性要求。Dynamo 在策略 3 下归档整个数据集很简单，因为分区的文件可以被分别归档。相反，在策略 1，Token 是随机选取的，归档存储的数据需要分别检索各个节点的 key，这通常是低效和缓慢的。策略 3 的缺点是，为维护分配所需的属性改变节点成员时需要协调，。

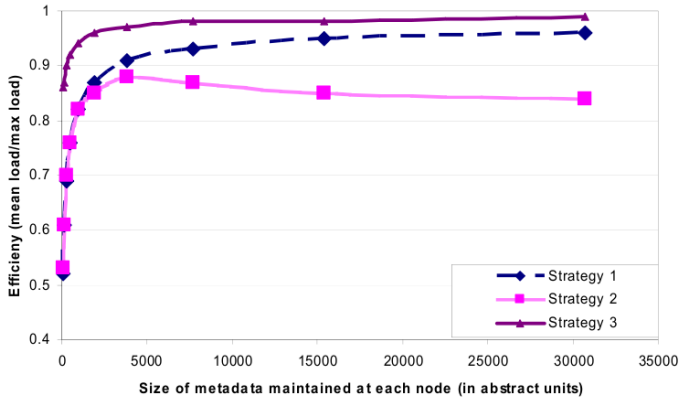

图 8：比较 30 个维持相同数量的元数据节的点,N=3 的系统不同策略的负载分布效率。系统的规模和副本的数量的值是按照我们部署的大多数服务的典型配置。

6.3 不同版本：何时以及有多少？

如前所述，Dynamo 被设计成为获得可用性而牺牲了一致性。为了解不同的一致性失败导致的确切影响，多方面的详细的数据是必需的：中断时长，失效类型，组件可靠性，负载量等。详细地呈现所有这些数字超出本文的范围。不过，本节讨论了一个很好的简要的度量尺度：在现场生产环境中的应用所出现的不同版本的数量。

不同版本的数据项出现在两种情况下。首先是当系统正面临着如节点失效故障的情况下，数据中心的故障和网络分裂。二是当系统的并发处理大量写单个数据项，并且最终多个节点同时协调更新操作。无论从易用性和效率的角度来看，都应首先确保在任何特定时间内不同版本的数量尽可能少。如果版本不能单独通过矢量时钟在语法上加以协调，他们必须被传递到业务逻辑层进行语义协调。语义协调给服务应用引入了额外的负担，因此应尽量减少它的需要。

在我们的下一个实验中，返回到购物车服务的版本数量是基于 24 小时为周期来剖析的。在此期间，99.94%的请求恰好看到了 1 个版本。0.00057%的请求看到 2 个版本，0.00047%的请求看到 3 个版本和 0.00009%的请求看到 4 个版本。这表明，不同版本创建的很少。

经验表明，不同版本的数量的增加不是由于失败而是由于并发写操作的数量增加造成的。数量递增的并发写操作通常是由忙碌的机器人(busy robot-自动化的客户端程序)导致而很少是人为触发。由于敏感性，这个问题还没有详细讨论。

6.4 客户端驱动或服务器驱动协调

如第 5 条所述，Dynamo 有一个请求协调组件，它使用一个状态机来处理进来的请求。客户端的请求均匀分配到环上的节点是由负载平衡器完成的。Dynamo 的任何节点都可以充当一个读请求协调员。另一方面，写请求将由 key 的首选列表中的节点来协调。此限制是由于这一事实——这些首选节点具有附加的责任：即创建一个新的版本标识，使之与写请求更新的版本建立因果关系(译：呜呜，这个很难！Causally subsumes)。请注意，如果 Dynamo 的版本方案是建基于**物理时间戳**(译：一个在本文中没解释的概念：[*Timestamp* Semantics and Representation]Many database management systems and operating systems provide support for time values. This support is present at both the logical and physical levels. The logical level is the user's view of the time values and the query level operations permitted on those values, while the physical level concerns the bit layout of the time values and the bit level operations on those values. The physical

level serves as a platform for the logical level but is inaccessible to the average user.)的话，任何节点都可以协调一个写请求。

另一种请求协调的方法是将状态机移到客户端节点。在这个方案中，客户端应用程序使用一个库在本地执行请求协调。客户端定期随机选取一个节点，并下载其当前的 Dynamo 成员状态视图。利用这些信息,客户端可以从首选列表中为给定的 key 选定相应的节点集。读请求可以在客户端节点进行协调，从而避免了额外一跳的网络开销(network hop)，比如，如果请求是由负载平衡器分配到一个随机的 Dynamo 节点，这种情况会招致这样的额外一跳。如果 Dynamo 使用基于时间戳的版本机制，写要么被转发到在 key 的首选列表中的节点，也可以在本地协调。

一个客户端驱动的协调方式的重要优势是不再需要一个负载平衡器来均匀分布客户的负载。公平的负载分布隐含地由近乎平均的分配 key 到存储节点的方式来保证的。显然，这个方案的有效性是信赖于客户端的成员信息的新鲜度的。目前客户每 10 秒随机地轮循一 Dynamo 节点来更新成员信息。一个基于抽取(pull)而不是推送(push)的方被采用，因为前一种方法在客户端数量比较大的情况下扩展性好些，并且服务端只需要维护一小部分关于客户端的状态信息。然而，在最坏的情况下，客户端可能持有长达 10 秒的陈旧的成员信息。如果客户端检测其成员列表是陈旧的(例如，当一些成员是无法访问)情况下，它会立即刷新其成员信息。

表 2 显示了 24 小时内观察到的，对比于使用服务端协调方法，使用客户端驱动的协调方法，在 99.9 百分位延时和平均延时的改善。如表所示，客户端驱动的协调方法，99.9 百分位减少至少 30 毫秒的延时，以及降低了 3 到 4 毫秒的平均延时。延时的改善是因为客户端驱动的方法消除了负载平衡器额外的开销以及网络一跳，这在请求被分配到一个随机节点时将导致的开销。如表所示，平均延时往往要明显比 99.9 百分位延时低。这是因为 Dynamo 的存储引擎缓存和写缓冲器具有良好的命中率。此外，由于负载平衡器和网络引入额外的对响应时间的可变性，在响应时间方面，99.9th 百分位这这种情况下(即使用负载平衡器)获得好处比平均情况下要高。

表二：客户驱动和服务器驱动的协调方法的性能。

	99.9th 百分读延时(毫秒)	99.9th 百分写入延时(毫秒)	平均读取延时时间(毫秒)	平均写入延时(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

6.5 权衡后台和前台任务

每个节点除了正常的前台 put/get 操作，还将执行不同的后台任务，如数据的副本的同步和数据移交(handoff)(由于暗示(hinting)或添加/删除节点导致)。在早期的生产设置中，这些后台任务触发了资源争用问题，影响了正常的 put 和 get 操作的性能。因此，有必要确保后台任务只有在不会显著影响正常的关键操作时运行。为了达到这个目的，所有后台任务都整合了管理控制机制。每个后台任务都使用此控制器，以预留所有后台任务共享的时间片资源(如数据库)。采用一个基于对前台任务进行监控的反馈机制来控制用于后台任务的时间片数。

管理控制器在进行前台 put/get 操作时不断监测资源访问的行为，监测数据包括对磁盘操作延时，由于锁争用导致的失败的数据库访问和交易超时，以及请求队列等待时间。此信息是用于检查在特定的后沿时间窗口延时(或失败)的百分位是否接近所期望的阀值。例如，背景控制器检查，看看数据库的 99 百分位的读延时(在最后 60 秒内)与预设的阈值(比如 50 毫秒)的接近程度。该控制器采用这种比

较来评估前台业务的资源可用性。随后，它决定多少时间片可以提供给后台任务，从而利用反馈环来限制背景活动的侵扰。请注意，一个与后台任务管理类似的问题已经在[4]有所研究。

6.6 讨论

本节总结了在实现和维护 Dynamo 过程中获得的一些经验。很多 Amazon 的内部服务在过去二年中已经使用了 Dynamo，它给应用提供了很高级别的可用性。特别是，应用程序的 99.9995%的请求都收到成功的响应(无超时)，到目前为止，无数据丢失事件发生。

此外，Dynamo 的主要优点是，它提供了使用三个参数的(N，R，W)，根据自己的需要来调整它们的实例。不同于流行的商业数据存储，Dynamo 将数据一致性与协调的逻辑问题暴露给开发者。开始，人们可能会认为应用程序逻辑会变得更加复杂。然而，从历史上看，Amazon 平台都为高可用性而构建，且许多应用内置了处理不同的失效模式和可能出现的不一致性。因此，移植这些应用程序到使用 Dynamo 是一个相对简单的任务。对于那些希望使用 Dynamo 的应用，需要开发的初始阶段做一些分析，以选择正确的冲突的协调机制以适当地满足业务情况。最后，Dynamo 采用全成员(full membership)模式，其中每个节点都知道其对等节点承载的数据。要做到这一点，每个节点都需要积极地与系统中的其他节点 Gossip 完整的路由表。这种模式在一个包含数百个节点的系统中运作良好，然而，扩展这样的设计以运行成千上万节点并不容易，因为维持路由表的开销将随着系统的大小的增加而增加。克服这种限制可能需要通过对 Dynamo 引入分层扩展。此外，请注意这个问题正在积极由 O(1)DHT 的系统解决(例如，[14])。

7 结论

本文介绍了 Dynamo，一个高度可用和可扩展的数据存储系统，被 Amazon.com 电子商务平台用来存储许多核心服务的状态。Dynamo 已经提供了所需的可用性和性能水平，并已成功处理服务器故障，数据中心故障和网络分裂。Dynamo 是增量扩展，并允许服务的拥有者根据请求负载按比例增加或减少。Dynamo 让服务的所有者通过调整参数 N,R 和 W 来达到他们渴求的性能，耐用性和一致性的 SLA。

在过去的一年生产系统使用 Dynamo 表明，分散技术可以结合起来提供一个单一的高可用性系统。其成功应用在最具挑战性的应用环境之一中表明，最终一致性的存储系统可以是一个高度可用的应用程序的构建块。

鸣谢

作者在此要感谢 PatHelland，他贡献了 Dynamo 的初步设计。我们还要感谢 MarvinTheimer 和 RobertvanRenesse 的评注。最后，我们要感谢我们的指路人(shepherd)，JeffMogul,他的详细的评注和 input(不知道怎样说了？词穷)在准备 camera ready 版本时，大大提高了本文的质量。