Dynamo: Amazon’s Highly Available Key-value Store

《Dynamo: Amazon’s Highly Available Key-value Store》这篇发表于07年，虽然时间久远但仍是一篇值得一读的文章，当然现在 Amazon 的 DynamoDB跟这篇文章介绍的架构应该是完全不一样了。感兴趣的同学可以再去读一下《Amazon DynamoDB: A Scalable, Predictably Performant, and Fully Managed NoSQL Database Service》

摘要

本文介绍了Dynamo的设计和实现，Dynamo是一个高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。为了达到这种级别的可用性，Dynamo在某些特定的故障场景下牺牲了一致性。Dynamo 使用了**对象版本化**（object versioning）和**应用协助的冲突解决**（application-assisted conflict resolution）机制，给开发者提供了一种新颖的接口。

1.引言

Amazon平台在性能、**可靠性**和效率等方面有着很高的要求，同时为了支持持续增长，平台需要具有高度的**可扩展性**。

系统的**可靠性**和**可扩展性**取决于如何管理它的应用状态。Amazon使用由数百个服务组成的高度去中心化、松耦合、面向服务的架构。在这样的环境中，特别需要永远可用的存储技术 。

任何时刻都会有比例小但数量不少（small but significant number）的服务器和网络设备出现故障。因此，软件系统要将故障视为正常的、可预测的行为（treat failure handling as the normal case），不影响可用性和性能。

Dynamo 用于管理对可靠性要求非常高的服务的状态，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。

Amazon平台上有许多服务只需要主键访问数据存储。例如畅销书列表、购物车、客户偏好、会话管理、销售排名和产品目录， 使用关系型数据库会导致低效，而且限制了规模的扩展性和可用性。Dynamo提供了一个 简单的主键接口来满足这些应用程序的需求。

Dynamo 基于一些熟知的技术实现了可扩展性和可用性：

- 通过**一致性哈希**对数据进行分区和复制（partitioned and replicated）
- 通过**对象版本化**（object versioning）实现一致性
- 副本之间的一致性由一种**类似仲裁的技术**（quorum-like technique）和一个去中心化的**副本同步协议**（replica synchroni protocol）保证

- **gossip-based** 分布式故障检测和成员检测协议

Dynamo 是一个完全去中心化的系统，很少需要人工管理。可以在Dynamo中添加和删除存储节点，而不需要任何手动分区或重新分配。

主要贡献:

- 评估如何将不同的技术结合起来提供一个单一的高可用性系统
- 证明了最终一致的存储系统可以用于要求苛刻的生产应用程序
- 还提供了对这些技术进行调优以满足具有非常严格的性能要求的生产系统的需求的见解。

2.背景

2.1 系统假设与需求

Query Model

通过唯一的 **key** 对数据进行读写（key-value store）。状态存储为二进制对象，以唯一的 key 索引。没有跨多个数据项的操作，也不需要关系模式（大部分服务可以使用这个简单的查询模型）。目标是需要存储相对较小的对象的应用程序 (通常小于1 MB)。

ACID

ACID（*Atomicity, Consistency, Isolation,Durability*）是保证数据库事务可靠执行的特性，在数据库中，对数据的单次逻辑操作称为一次事务。经验表明，提供ACID保证的数据存储的可用性往往是较差的。Dynamo 牺牲了 C (ACID中的“C”)，用较弱的一致性换来了高可用性。Dynamo 不提供任何隔离性的保证，只允许单个键的更新。

Efficiency

系统需要运行在通用硬件上。Amazon 的服务对延迟有着严格的要求，通常用百分位值P99.9衡量。同时存储系统必须满足那些严格的 SLA。另外，服务必须能够配置 Dynamo，使它们始终如一地实现其延迟和吞吐量的要求，权衡性能、成本效率、可用性和持久性。

Other Assumptions

Dynamo 只用在 Amazon 内部服务使用，假设其运行环境是非敌对的（non-hostile），不存在认证、授权等安全相关需求。因此可以不考虑安全性。此外，每个服务使用其不同的 Dynamo 实例，因此其初始设计目标是多达数百个存储主机的规模。

2.2 服务等级协议（SLA）

要保证一个应用完成请求所花的时间有一个上限，它所依赖的那些服 务就要有一个更低的上限。对于给定的系统特性，其中最主要的是客户端期望的请求率分布，客户端和服务端会定义一个 **SLA**（服务等级协议）来达成一致。

> SLA（Service-Level Agreement）服务等级协议，是指系统服务提供者（Provider）对客户（Customer）的一个可量化的服务承诺，常见于大型分布式系统中，用于衡量系统服务是否稳定健康的常见方法。

SLA 是一种服务承诺，因此指标具备多样性，业界主流常用指标包含：可用性、准确性、系统容量和延迟。举一个简单的SLA例子：一个服务，它保证在每秒500个请求的峰值客户端负载下，它将在300毫秒内为99.9%的请求提供响应。

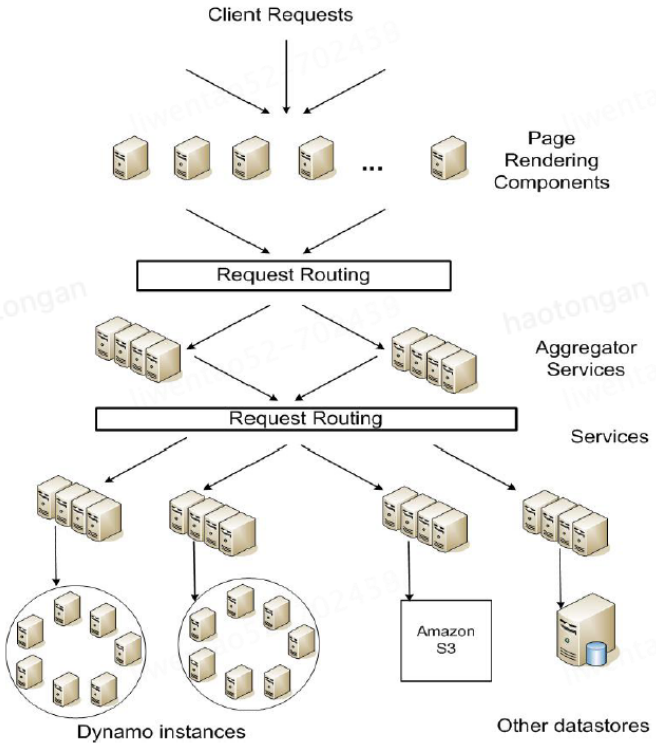


Figure 1: Service-oriented architecture of Amazon's platform

在 Amazon 去中心化的、面向服务的基础设施中，SLA 扮演着重要的角色。例如：购物网站的每个页面的渲染通常会涉及到上百个服务。为了保证用户体验，必须对每个服务的延迟做严格限制。

业界形成面向性能的 SLA 的常用方法是使用平均、中位数和预期方差来描述它。在 Amazon，我们发现，如果目标是建立一个让所有客户都有良好体验的系统，而不仅仅是大多数客户，那么这些指标就不够好。Dynamo 使用了 P99.9 分布。这个精度是经过大量实验分析，权衡了成本和性能之后得到的。 我们在生产环境的实验显示，这比基于均值或中位数的 SLA 有更好的用户体验。

存储系统通常在建立服务的 SLA 中扮演重要角色，特别是当业务逻辑相对于轻量级时，就像许多 Amazon 服务的情况一样。状态管理随后成为服务的 SLA 的主要组件。Dynamo 的主要设计考虑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

商业系统中传统的数据复制算法为了提供强一致性的数据接口，往往牺牲了在某些故障情况下的数据可用性。强一致性与高可用性在网络故障下不能同时得到满足。

为了提升系统在服务器和网络故障时的可用性，可以采用乐观复制（optimistic replication），异步地进行副本同步，并容忍并发操作和断联等。但这种方法可能会导致数据冲突，需要检测并解决冲突。但同时解决冲突的过程引入了两个问题：**什么时候解决**，**谁来解决**。Dynamo 被设计成一个最终一致的数据存储，也就是说所有的更新最终都会到达所有的副本。

when resolve

许多传统的数据存储在写过程中解决冲突，来保持简单的读复杂度，如果数据存储无法在给定时间到达所有（或大部分）的副本，则写操作可能会被拒绝。Dynamo 的设计目标是一个“始终可写”的数据存储（即数据存储高度可写），这个需求迫使我们将冲突解决的复杂性推到读操作上，以确保写操作永远不会被拒绝。

who resolve

可以由数据存储和应用程序来解决冲突。数据存储侧可以使用简单的策略，如“last write wins”等来解决。另一方面，由于应用程序知道数据模式，因此它可以决定最适合其客户体验的冲突解决方法。例如购物车的应用程序可以选择“合并”冲突的版本数据，并返回一个统一的结果。

设计中包含的其他关键原则：

Incremental scalability

支持节点扩展，同时对系统和运维的营销尽可能减少

Symmetry

Dynamo 中每个节点的职责应该是相同的，没有特殊节点或者说没有节点承担特殊责任或特殊角色（no master）。对称性简化了系统的配置和运维过程。

Decentralization

去中心化是对称性的扩展，设计应该支持去中心化的、点对点的（peer-to-peer），而不是集中的控制。这使得系统更具扩展性和可用性。

Heterogeneity

系统需要能够利用其运行的基础设施中的异构性。例如，工作负载必须与单个节点的能力成比例。这对于添加具有更大存储空间和处理能力的新节点而无需立即升级所有节点来说是非常重要。

3.相关工作

3.1 点对点系统

一些点对点（peer-to-peer, P2P）系统关注了数据存储和分散（data storage and distribution）的问题。

第一代 P2P 系统如 Freenet 和 Gnutella，在文件共享方面广泛应用，它们建立在非受信的网络上，通过随机的覆盖网络链路连接节点，并使用泛洪查询来查找数据。

随后，发展出结构化 P2P 网络，这种网络利用全局一致性协议高效地路由查询到存储特定数据的节点。系统如 Pastry 和 Chord 通过路由机制保证了查询在有限跳数内得到响应。为了减少路由带来的延迟，一些系统采用了 O(1) 路由机制，使得节点可以在常量跳数内将请求路由到目标节点。

在这些路由覆盖之上，构建了如 Oceanstore 和 PAST 等存储系统。Oceanstore 提供了全球分布式、事务型、持久的存储服务，并采用冲突解决的方法来处理并发更新，从而避免了广域锁带来的问题。它按顺序应用一组更新作为原子操作到所有副本上，适用于在不受信基础设施上的数据复制。

相比之下，PAST 是建立在 Pastry 之上的一个简单抽象层，它提供了持久和不可变对象的存储，并假设应用可以在其上构建所需的存储语义，如可变文件。

3.2 分布式文件系统与数据库

文件系统和数据库系统领域已经对通过分发数据（distributing data）来获得性能、可用性和持久性已经进行了广泛研究。与只支持扁平命名空间（flat namespaces）的 P2P 存储系统相比，分布式文件系统通常支持层级命名空间（hierarchical namespaces）。

Ficus 和 Coda 这样的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 解决过程进行管理。Farsite系统是一个分布式文件 系统，它不使用任何中心化服务器（像NFS），Farsite 使用复制实现高可用性和可伸缩性。Google File System 是为托管 Google 内部应用状态而构建的另一个分布式文件系统。GFS 使用简单的设计，用一个主服务器来托管整个元数据，其中数据被分成块并存储在不同的块服务（chunkservers ）中。Bayou 是一个分布式的关系型数据库系统，它允许失联操作（disconnected operations），并提供最终一致性。

3.3 讨论

Dynamo 与上述去中心化的存储系统的不同之处在于它的目标需求。

- Dynamo 被设计为需要“始终可写”数据存储的应用程序，其中不会因失败或并发写而拒绝更新数据。
- Dynamo 是为单个管理域中的基础设施构建的，其中假定所有节点都是受信任的。
- 使用 Dynamo 的应用程序不需要支持分层命名空间或复杂的关系模式。
- Dynamo 是为对延迟敏感的应用程序而构建的，这些应用程序要求至少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，Dynamo 可以表征为零跳分布式哈希表（zero-hop DHT），其中每个节点在本地维护足够的路由信息，以便将请求直接路由到适当的节点。

4.系统架构

生产环境中运行的存储系统架构是比较复杂。除了实际的数据持久组件外，系统还需要具有可扩展和健壮的解决方案：用于负载均衡（load balancing）、成员管理（membership）和故障检测（failure detection）、故障恢复（failure recovery）、副本同步（replica synchronization）、过载处理（overloadhandling）、状态转移（state transfer）、并发（concurrency）和作业调度（job scheduling）、请求编组（request marshalling）、请求路由（request routing）、系统监控（system monitoring）、 告警（alarming）、 和配置管理（configuration management）。

本文主要关注 Dynamo 中使用的核心分布式系统技术：

- 分区（partitioning）
- 复制（replication）
- 版本化（versioning）
- 成员管理
- 故障处理
- 规模扩展（scaling）

Table 1: Summary of techniques used in *Dynamo* and their advantages.

Problem	Technique	Advantage
Partitioning	Consistent Hashing	Incremental Scalability
High Availability for writes	Vector clocks with reconciliation during reads	Version size is decoupled from update rates.
Handling temporary failures	Sloppy Quorum and hinted handoff	Provides high availability and durability guarantee when some of the replicas are not available.
Recovering from permanent failures	Anti-entropy using Merkle trees	Synchronizes divergent replicas in the background.
Membership and failure detection	Gossip-based membership protocol and failure detection.	Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information.

Table 1 presents a summary of the list of techniques Dynamo uses and their respective advantages.

4.1 系统接口

Dynamo 通过简单的接口存储与键相关的对象，它暴露两个操作 `get()` 和 `put()` 。

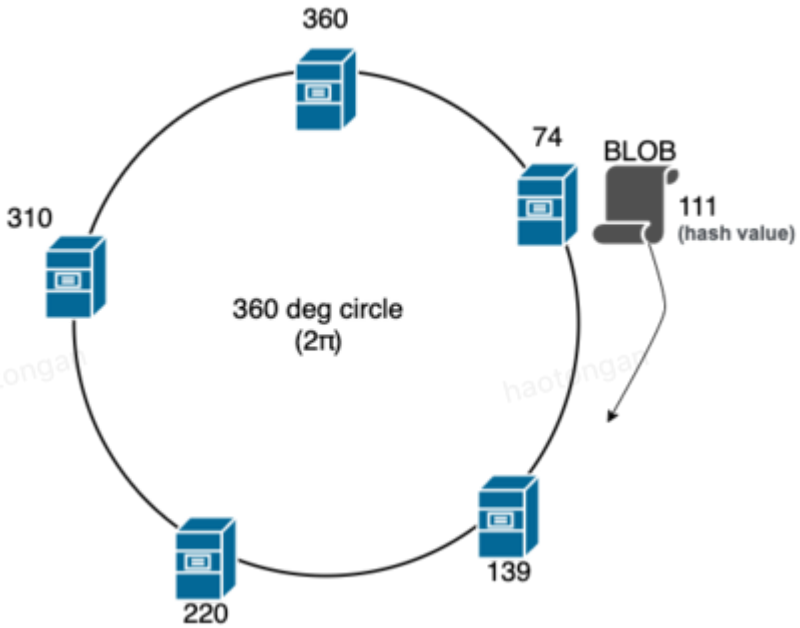
`get(key)` 操作会在存储系统中定位与该键关联的对象副本，并返回单个对象或具有冲突版本的对象列表以及上下文（context） `put(key, context, object)` 操作会根据关联的键确定对象的副本应该放在哪里，并将副本写入磁盘。上下文编码了系统中对象的元数据，包含了对象版本等信息，对调用者是不透明的（opaque）。上下文是和对象存储在一起的，系统可以验证 put 请求中提供的上下文对象的有效性。

Dynamo 将调用者提供的键和对象都视为不透明的字节数组。对 key 使用 MD5 以生成128位标识符，该标识符用于确定存储节点。

4.2 分区算法

Dynamo 一个关键设计要求是它必须支持增量扩展，这就需要一种机制来动态地在系统中的节点集合中对数据进行分区。Dynamo的分区方案采用于一致性哈希（consistent hashing）来在多个存储节点上进行负载分配。

> In computer science, consistent hashing is a special kind of hashing technique such that when a hash table is resized, only n/m keys need to be remapped on average where n is the number of keys and m is the number of slots. In contrast, in most traditional hash tables, a change in the number of array slots causes nearly all keys to be remapped because the mapping between the keys and the slots is defined by a modular operation



在一致性哈希中，哈希函数的输出范围被视为一个固定的循环空间或“环”，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”。由 key 标识的每个数据项都被分配给一个节点，方法是对数据项的 key 进行散列以产生其在环上的位置，然后顺时针走环以找到位置大于该项目位置的第一个节点。

因此，每个节点负责环中它与其在环上的前任节点之间的区域。 一致性哈希的主要优点是节点的离开或加入仅影响其直接邻居，而其他节点不受影响。

基本的一致性哈希算法存在一定的挑战：

- 环上每个节点的位置随机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构性。

Dynamo 使用了“虚拟节点”的概念。 虚拟节点看起来就像系统中的单个节点，但每个节点可以负责多个虚拟节点。每个节点都被分配到环中的多个点，而不是将节点映射到环中的单个点。 也就是说，当一个新节点添加到系统中时，它会在环中被分配多个位置（以下称为“令牌”）。

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到其余可用节点上。
- 当某个节点再次可用，或者向系统添加新节点时，新的可用节点会接受来自其他每个可用节点的大致相等的负载量。
- 节点负责的虚拟节点的数量可以根据其容量来决定，并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 在多个主机上复制其数据。 每个数据项都在 N 个主机上复制，其中 N 是“每个实例”配置的参数。 每个 key 都分配给一个协调者（coordinator）节点（上面一致性哈希分配的节点）。 coordinator 负责复制其范围内的数据项。 除了在本地存储其范围内的每个 key 之外，协调器还在环中的 N-1 个顺时针后继节点处复制这些 key。 这导致系统中每个节点负责它与其第 N 个前驱节点之间的环区域。

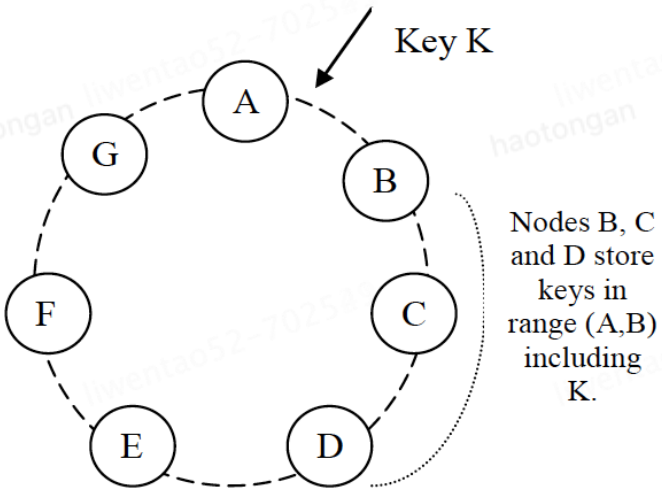


Figure 2: Partitioning and replication of keys in Dynamo ring.

如上图所示，节点 B 除在本地存储之外，还作为 coordinator 复制 key 至节点 C 和节点 D，节点 D 实际存储范围为(A, B]，(B, C]和(C, D]的 key。

负责存储某个特定 key 的节点列表称为偏好列表（preference list）。4.8节指出，系统的设计使得系统中的每个节点都可以确定对于任何特定的 key，哪些节点应该在此列表中。考虑到节点会存在故障，偏好列表可能包含超过N个节点。考虑到使用了虚拟节点，特定 key 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 个位置中的多个位置）。为了解决这个问题，通过跳过环中的位置来构造key 的偏好列表，以确保该列表仅包含不同的物理节点。

4.4 数据版本

Dynamo 提供最终一致性，允许更新异步地传递给所有的副本。`put()` 操作允许更新应用到所有副本上之前就给予调用返回，这可能导致后续 `get()` 操作可能无法返回最新的数据。如果更新过程没有发生失败，那么 更新传播的时间就有一个界限。但是，在某些故障场景下(例如，服务器中断或网络分区)，更新可能在很长一段时间内无法到达所有副本。

Amazon 有一类应用程序可以容忍这种不一致，在这些情况下可以继续运行。以购物车应用为例，“添加到购物车”的请求永远不能被丢失或拒绝。如果购物车最新的状态不可用，那么，对一个旧版本的购物车数据进行了修改也是有意义的，需要保留。但是这不能直接覆盖最新状态，因为最新状态中有一些修改也需要进行保留。当客户希望将商品添加到(或从)购物车中删除时，如果最新的版本不可用，则将该商品添加到(或从) (较旧的版本)，并稍后协调不同的版本。

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出现在系统中。

> Dynamo treats the result of each modification as a new and immutable version of the data

冲突协调方式

- 基于句法的协调（syntactic reconciliation）
- 基于语义的协调（semantic reconciliation）

大多数情况下，新版本包含了以前的版本，系统本身可以确定权威版本（句法协调）。但是，如果存在失败和并发更新，则可能发生版本分支，从而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，以便将数据演变的多个分支合并回一个（语义协调）。

vector clock

Dynamo 使用向量时钟**（vector clock）**来记录同一对象的不同版本之间的因果关系。

> A vector clockis adata structure used for determining thepartial orderingof events in adistributed systemand detecting causality violations. Just as in Lamport timestamps, inter-process messages contain the state of the sending process'slogical clock. A vector clock of a system of N processes is an array/vector of N logical clocks, one clock per process; a local "largest possible values" copy of the global clock-array is kept in each process.

一个向量时钟就是一个 `(node, counter)` 列表。一个向量时钟关联了一个对象的所有版本，可以通过向量时钟来判断对象的两个版本是否在并行的分支上，还是具有因果顺序的。如果第一个对象时钟上的counter 小于或等于第二个时钟中的所有 counter，则第一个对象是第二个对象的祖先（ancestor），可以被忽略或删除。否则，这两个修改是冲突的，需要协调合并。

如果A1[<node1, 1>]，A2[<node1,1>,<node2,2>]，那么A1是A2的祖先，两者存在因果顺序。A1可以被忽略或删除。如果A1[<node1, 1>]，B1[<node2,1>]，则两个版本存在冲突。

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。这是通过传递它从先前的读取操作获得的 context 来实现的，context 中包含向量时钟的信息。

在处理读请求的时候，如果 Dynamo 能够访问到多个分支版本，并且无法协调（syntactically reconciled），那它就会返回所有版本的对象，并在 context 中附带各自的向量时钟信息。基于 context 的更新认为协调了版本（semantic reconciliation），解决了冲突，将多个分支重新合并为一个新分支。

下面来看一下论文中的例子

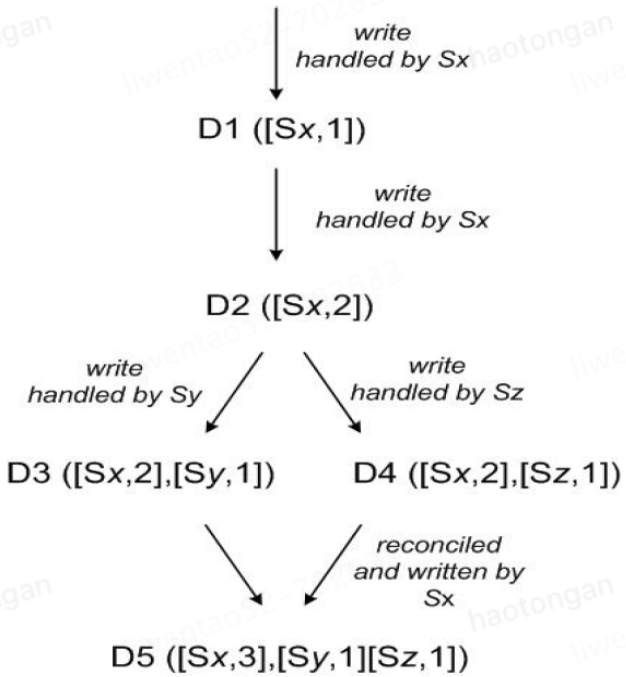


Figure 3: Version evolution of an object over time.

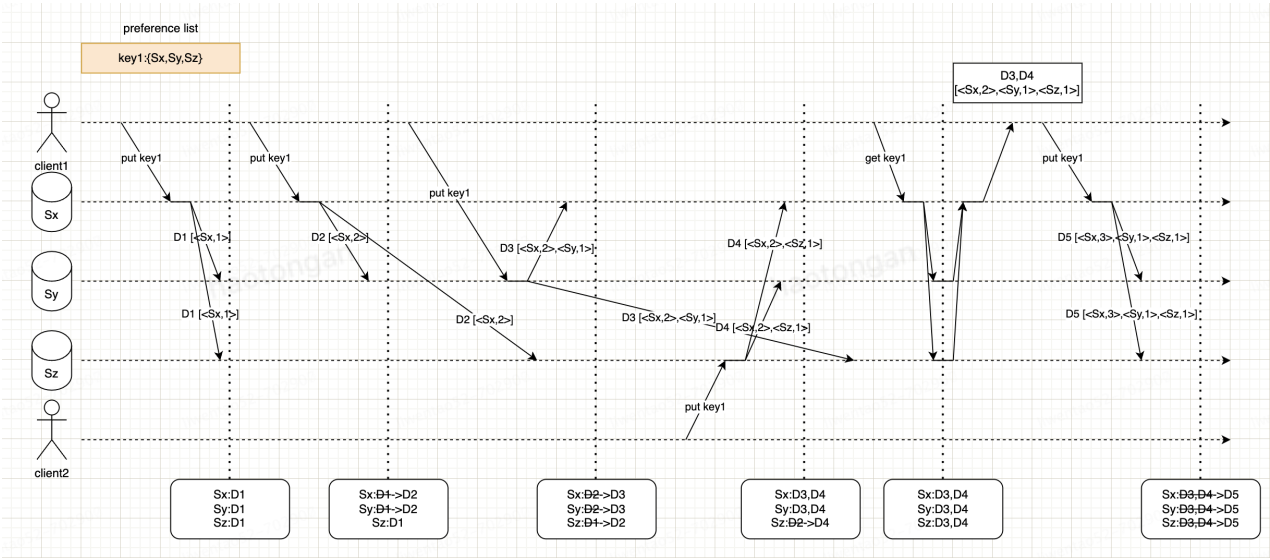
客户端写一个新对象。处理该 key 写操作的节点（Sx）增加其序列号（counter），并使用它来创建数据的向量时钟。那么现在，系统有对象 D1 和它关联的时钟：[<Sx, 1>]。

客户端更新这个对象。假设同样的节点（Sx）也处理这个请求。系统现在还有对象 D2 及其关联时钟：[<Sx, 2>]。D2 来自于 D1，因此 D1 可以被覆盖掉。但是 D2 的这次更新可能还没同步到其他节点上，其他节点上可能还是只有 D1 的信息。

假设相同的客户端再次更新了对象，而这次由另一个节点（Sy）处理请求。系统现在有数据 D3 和它的 关联时钟：[<Sx, 2>，<Sy, 1>]。

接下来，假设另一个客户端读取 D2 并尝试更新它，另一个节点（Sz）处理这次写入操作。系统现在有D4 (D2的后代)，其时钟为：[<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删除。 一个知道 D3 并接收到 D4 的节点会发现它们之间不存在因果关系。 换句话说，D3 和 D4 的改动没有反映在对方之中。 这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读取了 D3 和 D4（context 将反映读取找到了这两个值）。读取的 context 包含了 D3 和 D4 的向量时钟，即[<Sx, 2>，<Sy, 1>, <Sz, 1>]。 如果客户端执行协调并且由节点 Sx 协调写入，Sx 将更新其在时钟中的序列号。 新数据 D5 将具有以下时钟：[<Sx, 3>，<Sy, 1>, <Sz, 1>]。



可能存在的问题

如果较多的 coordinator 节点写入对象，会使得向量时钟较长，但正如上面说到的，通常来讲都是由偏好列表中的前 N 个节点作为 coordinator 写入数据，因此不会过长。但由于可能存在网络分区或多个节点故障的情况出现，这种情况应该限制向量时钟的长度。为此，Dynamo 采用了以下时钟截断方案: 与每个（node，counter）对一起，Dynamo 存储一个时间戳，该时间戳表示节点最后一次更新数据项的时间。当节点对达到一个阈值时（10），就将最老的删除。

4.5 get () 和 put () 操作执行

Dynamo 中的任何存储节点都有资格接收任何 key 的客户端 get 和 put 操作。在本节中，将描述如何在无故障的环境中执行这些操作（下一节介绍有故障情况）。

客户端可以使用两种策略来选择节点

1. 通过一个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一个节点
2. 使用能分区感知的客户端，直接将请求路由到合适的 coordinator 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。

处理读或写操作的节点称为 coordinator。通常是偏好列表中前 N 个节点中的第一个。如果通过负载均衡器接收请求，则访问 key 的请求可能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 key 的偏好列表的前 N 个中，则该节点不会处理该请求。 相反，该节点会将请求转发到偏好列表中前 N 个节点中的第一个。

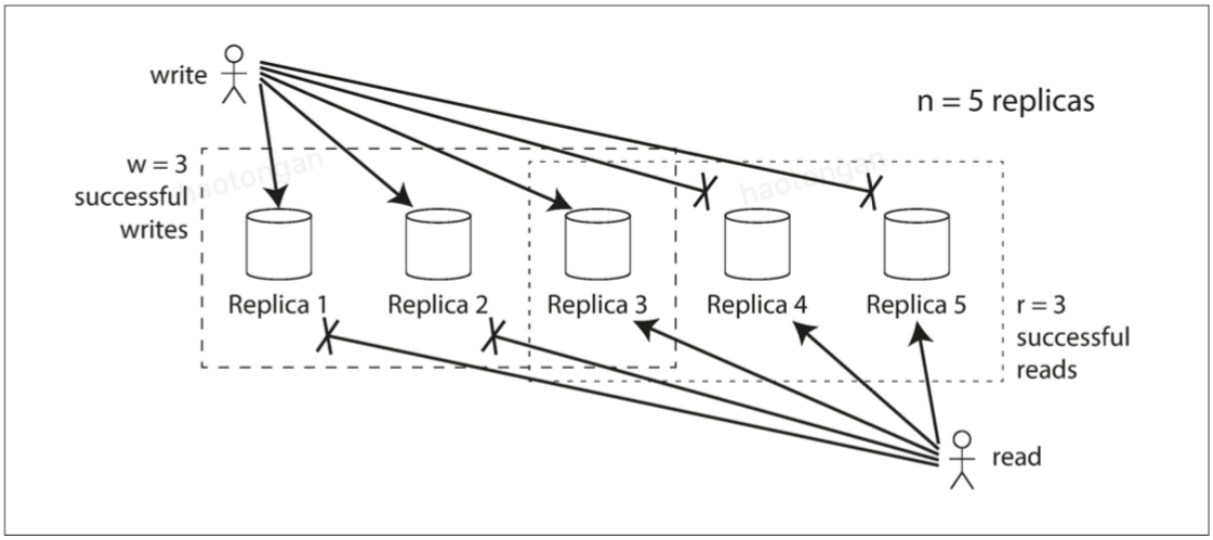
读/写操作通常涉及到偏好列表中的前 N 个节点（跳过那些宕机或者无法访问的节点）。如果所有的节点都是健康的，前 N 个节点将会被访问，如果出现了节点故障或网络分区的情况，偏好列表中排序较低的节点将会被访问。

quorum

为了保持副本之间的一致性，Dynamo 使用类似于 quorum systems 中使用的一致性协议。该协议有两个关键可配置值 R 和 W。

- R:参与成功读操作所需的最小节点数
- W:参与成功写操作所需的最小节点数

quorum systems 通常设置 R 和 W 使得 R + W > N （N 为分布式系统中节点的个数）。 在此模型中，get/put 操作的延迟由最慢的 R/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。



- 如果 W < N，当节点不可用时，我们仍然可以处理写入。
- 如果 R < N，当节点不可用时，我们仍然可以处理读取。
- 对于 N=3，W=2，R=2，我们可以容忍一个不可用的节点。
- 对于 N=5，W=3，R=3，我们可以容忍两个不可用的节点。 这个案例如上图所示。
- 通常，读取和写入操作始终并行发送到所有 N 个副本。参数 W 和 R 决定我们等待多少个节点，即在我们认为读或写成功之前，有多少个节点需要返回成功。

Dynamo 中在收到对 key 的 put 请求后，coordinator 会生成新版本的向量时钟并在本地写入新版本。 然后，coordinator 将新版本（以及新的向量时钟）发送到 N 个排名最高的可到达节点。 如果至少有 W-1 个节点响应则认为该次写入是成功的。

对于 get 请求也是类似的，coordinator 从该 key 的偏好列表中前 N 个健康节点请求该 key 的所有数据版本，然后等待 R 个响应，然后将结果返回给客户端。 如果 coordinator 最终收集了数据的多个版本，它将返回它认为无因果关系的所有版本。 然后客户端协调不同的版本，并将最新的版本数据写回。

4.6 故障处理：间接移交（Handling Failures: Hinted Handoff）

如果 Dynamo 采用传统的 quorum 机制，那么当服务出现故障或者网路分区时无法保持可用，即便是最简单的故障出现时，持久性也会降低。

为了解决这个问题，Dynamo 并不严格执行 quorum 机制，而是采用一个较为宽松的 quorum 机制（sloppy quorum），也就上节提到的，偏好列表中前 N 个节点负责处理读写操作（实际分布式系统中存储的节点数量时要超过 N 的，所以称其为 sloppy quorum）。当然也不一定就是哈希还中的前 N 个节点，因为可能会出现节点不可用或网络分区，此时要跳过该问题节点。

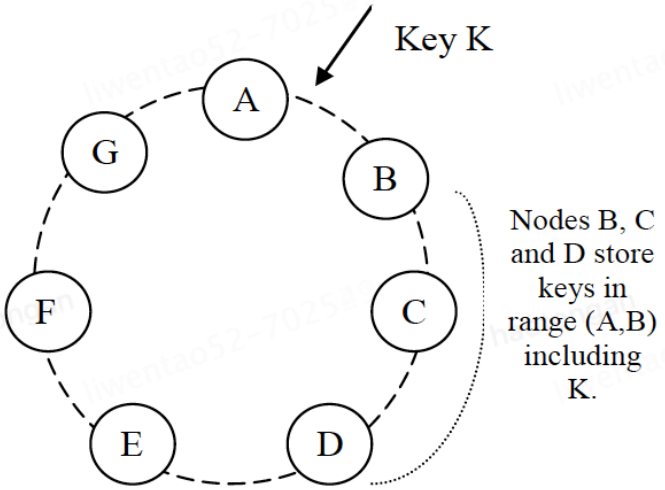


Figure 2: Partitioning and replication of keys in Dynamo ring.

以上图为例，N=3，如果节点 A 在写入操作期间暂时故障或无法访问，则通常本应发送给节点 A 上的副本会发送到节点 D。这样做的目的是为了维持可用性和持久性。 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，本例即为节点 A。 接收提示副本的节点会将它们保存在定期扫描的单独本地数据库中。 一旦检测到节点 A 恢复，节点 D 将尝试将副本传送给节点 A。一旦传送成功，节点 D 就可以从其本地存储中删除该对象，而不会减少系统中的副本总数。

由于使用了该机制，Dynamo 可确保读/写操作不会因节点或网络临时出现故障而失败。 需要最高级别可用性的应用程序可以将 W 设置为1，这确保只要系统中的一个节点将 key 持久写入成功，写入就会被系统所接受。 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，大多数生产服务设置了更高的W来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。

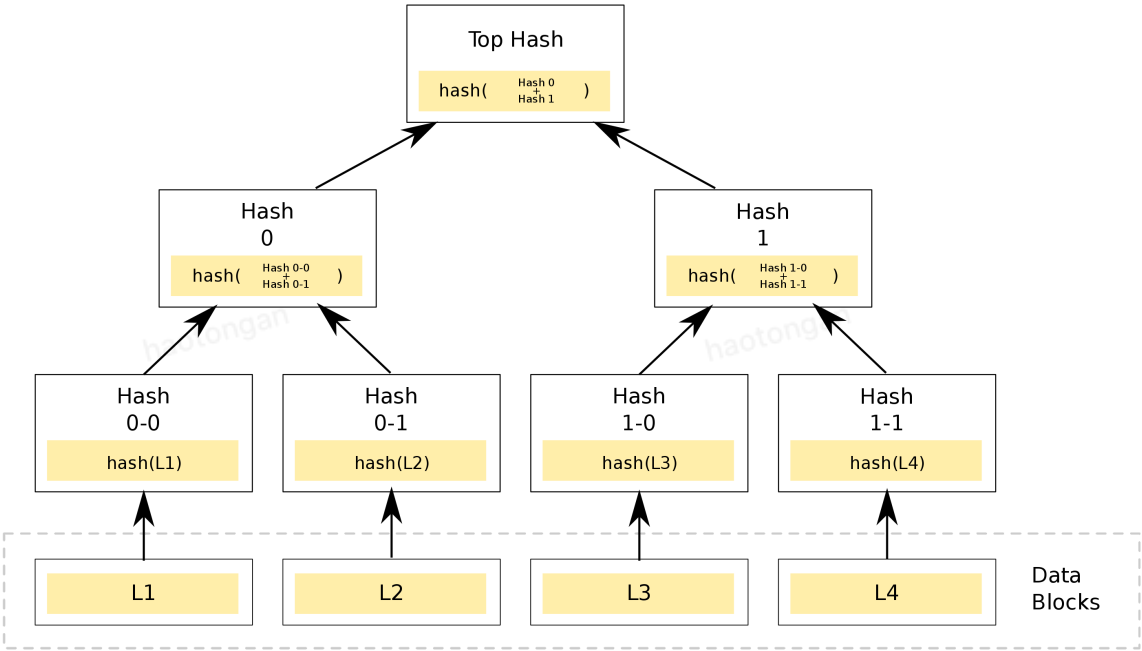
高度可用的存储系统必须能够处理整个数据中心的故障 。Dynamo的配置使得每个对象都跨多个数据中心复制。实质上，key 的偏好列表的构造使得存储节点分布在多个数据中心。这些数据中心通过高速网络链路连接，在整个数据中心挂掉的情况下仍然可以提供服务。

4.7 永久故障处理：副本同步

如果系统节点变动不频繁并且节点故障是暂时性的，则 Hinted Handoff 效果很好。在某些情况下，执行 hinted 的节点在副本返回到原始副本节点就不可用了（上节，在 A 恢复前，D 先不可用了）。为了处理这个问题和其他对持久性的威胁，Dynamo 实现了一个反熵（anti-entropy）副本同步协议来保证副本同步。

Merkle tree/Hash tree

为了更快地检测副本之间的不一致性并最小化传输的数据量，Dynamo 使用了 Merkle tree 这一数据结构,其中叶子是各个 key 对应 value 的哈希值。父节点则是子节点的哈希值。



Merkle tree 的主要优点：

- 树中的每个分支都可以独立检查 ，不需要节点下载整个树或整个数据集。
- 在检查副本之间的不一致性时，Merkle tree 可以减少数据传输量。

例如，如果两棵树的根的哈希值相等，则树中的叶节点的值相等，并且节点不需要同步。 如果不是，则意味着某些副本的值不同。 在这种情况下，节点（node）可以交换子节点的哈希值，并且该过程将继续，直到到达树的叶子，此时主机可以识别“不同步”的 keys。 Merkle tree 最大限度地减少了同步所需传输的数据量，并减少了反熵过程中执行的磁盘读取次数。

每个节点为其托管的每个 key 范围（虚拟节点覆盖的 key）维护一个单独的 Merkle tree。 这允许节点比较 key 范围内的 key 是否是最新的。 在此方案中，两个节点交换与它们共同托管的 key 范围相对应的 Merkle tree 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。 该方案的缺点是，当节点加入或离开系统时，许多 key 范围会发生变化，从而需要重新计算。 不过，这个问题已通过第 6.2 节中描述的细化分区方案得到解决。

4.8 成员资格与故障检测

4.8.1 Ring Membership

在 Amazon 的环境中，节点中断（故障或维护）通常是暂时的，但也可能会持续很长时间。一个节点服务中断并不能说明这个节点永久性的离开了系统，因此不应该导致分区再平衡，或者修复无法访问的副本。同样地，人工失误也可能无意中启动新的 Dynamo 节点。由于这些原因，使用显式机制（explicit mechanism）来对 Dynamo 环中节点进行添加和删除是合适的。 管理员使用命令行工具或浏览器连接到 Dynamo 节点并发出成员资格更改以将节点加入环或从环中删除。 服务请求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，因为节点可以多次删除和添加回来。 gossip-based 协议传播成员资格变更并维护成员资格的最终一致视图。 每个节点每秒都会联系一个随机选择的节点，并且两个节点有效地协调其持久化的成员资格更改历史。

当节点第一次启动时，它会选择其令牌集（一致哈希空间中的虚拟节点）并将节点映射到各自的令牌集。 该映射持久化保存在磁盘上，最初仅包含本地节点和令牌集。 存储在不同 Dynamo 节点的映射在交换成员资格变更历史期间进行协调。 因此，分区和数据放置信息也通过 gossip-based 协议传播，并且每个存储节点都知道其它节点处理的令牌范围。 这允许每个节点将 key 的读/写操作直接转发到正确的节点集。

4.8.2 External Discovery

上述机制可能暂时导致 ring 逻辑分区。例如，管理员可以先将节点 A 加入到环中，然后再将 B 加入到环中。在这种情况下，节点 A 和节点 B 都认为自己是环的成员，但是并不会立即感知到对方。为了避免逻辑分区，将一些 Dynamo 节点作为种子节点。种子节点是通过外部机制所发现的，所有节点都感知其存在。因为所有节点最终都会和种子节点协调成员信息，所以逻辑分区就几乎不可能发生。种子从静态配置文件中获取，或者从一个配置服务中获取。通常情况下，种子节点具有普通节点的全部功能。

4.8.3 Failure Detection

Dynamo 中的故障检测用于避免在读/写操作期间以及传输分区和 hinted 复制时尝试与无法访问的节点通信。为了避免尝试通信失败，一个本地的故障检测机制就足够了，即：如果节点 B 没有响应节点 A 的消息（即使节点 B 响应了节点 C 的消息），那么节点 A 可认为节点 B 不可访问。在客户端请求的稳定存在，此时Dynamo 环中节点之间通信也是正常的。当节点 B 无法响应消息时，节点 A 可以快速的发现节点 B 此时已经是无法访问的了，节点 A 会将本应发送给节点 B 的请求发送给备用节点（偏好列表）。同时定期检测节点 B 是否恢复。在没有客户端请求驱动两个节点之间的流量的情况下，节点之间不需要真正知道另一个是否可达和能否响应。

去中心化的故障检测协议使用简单的 gossip 风格协议，该协议使系统中的每个节点能够了解其他节点的 到达（或离开）。Dynamo 的早期设计使用这种协议来维护全局一致的故障状态视图。后来确定了显式机制来增加和删除节点不在需要故障状态全局视图了。

4.9 添加/移除存储节点

当一个新节点 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 负责的每个 key 范围（每个虚拟节点负责处理的那段范围），当前可能已经有多个节点（小于或等于 N）在负责处理了。由于将这些 key 范围分配给了节点 X，那么一些现有节点则不再需要处理这部分 key 范围了，这些节点将这些 key 转移到节点 X 中。

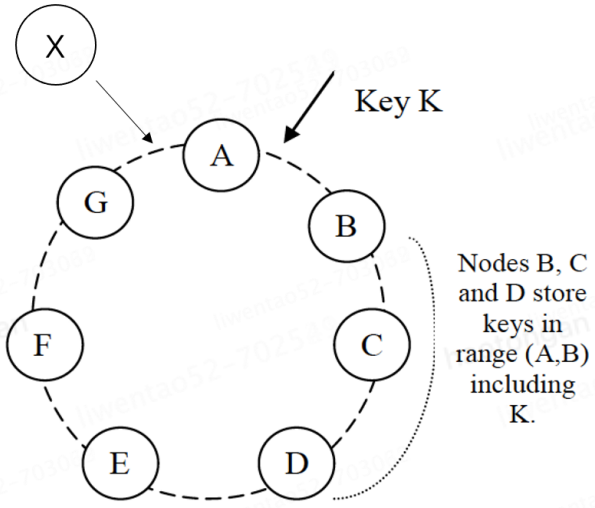


Figure 2: Partitioning and replication of keys in Dynamo ring.

考虑一个简单的场景，节点 X 加入到节点 A 和节点 B 之间。当节点 X 添加到系统中时，它负责存储 (F, G]、(G, A] 和 (A, X] 范围内的 key（N = 3）。那么，节点 B、C 和 D 不再需要将 key 存储在各自的范围中，也就是说节点 B 不再负责 (F, G]，节点 C 不再负责 (G, A]，节点 D 不再负责(A, X] 范围内的 key 了。 因此，节点 B、C 和 D 在得到节点 X 的确认后将向 X 传输适当的 key set。 当一个节点从系统中删除时，重新分配过程与上面描述的相反。

5.实现

在 Dynamo 中，每个存储节点都具有三个主要组件，均使用 Java 实现：

- 请求协调（request coordination）
- 成员资格和故障检测（membership and failuredetection）
- 本地持久化引擎（local persistence engine）

local persistence engine

Dynamo 的本地持久化组件允许插入不同的存储引擎。正在使用的引擎包括

- Berkeley Database (BDB) Transactional Data Store
- BDB Java Edition

- MySQL
- In-memory buffer with persistent backing store

设计可插拔持久化组件的主要原因是可以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 KB 数量级的对象，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 Transactional Data Store。

request coordination

request coordination 组件构建在基于事件驱动架构的 Messaging Substrate 之上，其中消息处理管道被分为多个阶段，类似于 SEDA 架构。所有通信都基于 Java NIO channel 实现。

客户端的读取和写入请求实际是通过 coordinator 来执行的，读操作时从一个或多个节点收集数据，写操作时同样在一个或多个节点存储数据。每个客户端请求都会在接收请求的节点上创建状态机。 状态机包含用于识别负责 key 的节点、发送请求、等待响应、重试、处理响应以及封装响应返回给客户端的所有逻辑。 每个状态机实例只处理一个客户端请求。

读操作实现以下状态机（省略了失败处理和重试的状态）

1. 向节点发送读请求
2. 等待返回请求最小响应数量（R）
3. 如果在给定的时间范围内收到的响应太少，则请求失败
4. 收集所有数据版本并确定要返回的版本
5. 如果启用了版本控制，则执行语法协调并生成一个context，包含所有剩余版本的向量时钟

在 coordinator 将读操作响应返回给客户端后，状态机还会等待一小段时间来接收那些未完成的响应（最小读取数量外的或超时响应的）。如果这些响应中存在过期的版本对象，那么 coordinator 会将最新的版本对象更新到这些节点，这一过程称为读修复，因为这是在取巧的时间（opportunistic time）修复了错过最近更新的副本（replicas that have missed a recent update），相当于替代了反熵协议同步数据。

写请求由偏好列表的前 N 个节点中的一个来协调处理，如果总是由第一个节点协调，虽然有一定的好处，比如写入操作可以保证一定的顺序，但是这会导致不均匀的负载分布，损害了 SLA。为了解决这个问题，允许偏好列表中前 N 个节点中的任何一个节点协调写入。通常，一个写操作之前还会有一个读操作，因此，前一次读操作返回最快的那个节点，来作为写操作的 coordinator ，这个信息存储在读操作返回的 context 中。这种优化能够选择具有前一次读取操作并协调写入的节点更容易被选中，从而增加 read-your-writes 数据一致的机会。 它还减少了请求处理的抖动，从而提高了P99.9的性能。

6.学习到的经验与教训

Dynamo 在不同类型的服务中，配置也不相同。主要体现在版本协调逻辑和读/写仲裁特点上，几种主要的模式如下。

Business logic specific reconciliation

这是 Dynamo 的一个流行用例。 每个数据对象都跨多个节点复制。 如果版本不同，客户端应用程序将执行自己的协调逻辑。 前面讨论的购物车服务就是一个典型例子。 其业务逻辑通过合并客户购物车的不同版本来协调对象

Timestamp based reconciliation

该模式与上面的协调机制不同，如果对象版本不同，Dynamo 会执行简单的基于时间戳的协调逻辑“last write wins”； 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是使用此模式的服务的一个很好的示例。

High performance read engine

虽然 Dynamo 被构建为“始终可写”的数据存储，但一些服务可以调整其仲裁特性并将其用作高性能读取引擎。 通常，这些服务读请求的频率远大于写。 在此配置中，通常 R 设置为 1，W 设置为 N。对于这些服务，Dynamo 提供跨多个节点分区和复制数据的能力，从而提供增量可扩展性。 其中一些实例在更重量级后备存储中作为权威持久性缓存来用于数据存储。 维护产品目录和促销品的服务属于此类模式。

Dynamo 的主要优点是，客户端应用程序可以调整 N、R 和 W 的值，以达到性能、可用性和持久性所需的等级。例如，N 的值决定了每个对象的持久性。Dynamo 用户通常使用的 N 是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，那么只要系统中只要有一个节点可以成功处理写请求，那么写请求就永远不会被拒绝。 但是，较低的 W 和 R 可能会增加不一致的风险，因为即使大多数副本未处理写入请求，写入请求也会被视为成功并返回给客户端。 这也会引入一个持久性上的隐患窗口（vulnerability window），即使它只在少数节点上进行了持久化，写请求也会成功返回客户端。

传统观点认为，持久性和可用性是携手并进的（go hand in-hand）。 然而这里不一定如此。 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概率（降低了可用性），因为更多的存储节点需要处于存活时才能处理写入请求。

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必要等级的 SLA，即性能、持久性、一致性和可用性。

本节中介绍的所有数据测量都是基于（3,2,2）配置的，并且在数百个具有同质硬件配置的节点的实时系统上进行的。Dynamo 中每个实例中的节点都跨多个数据中心部署的。这些数据中心通常通过高速网络链路连接。 回想一下，一次成功的读/写操作需要 R/W 个节点响应 coordinator。 显然，数据中心之间的网络延迟会影响响应时间，并且节点的选择（及其数据中心位置）应满足应用程序目标的SLA。

6.1 平衡性能与持久性

虽然 Dynamo 的主要设计目标是构建一个高可用的数据存储，但性能也是一个同样重要的指标。Amazon的服务性能通常使用P99.9或P99.99来衡量。使用 Dynamo 的服务的典型SLA要求是 P99.9 的读写请求在300ms内执行完成。

由于 Dynamo 运行在标准商用硬件设备上，其I/O吞吐量远低于高端企业服务器，因此提供一致的高性能读写服务是一项非常重要的任务。读写操作中涉及多个存储节点使其更具挑战性，因为这些操作的性能受到最慢的R或W副本的限制。

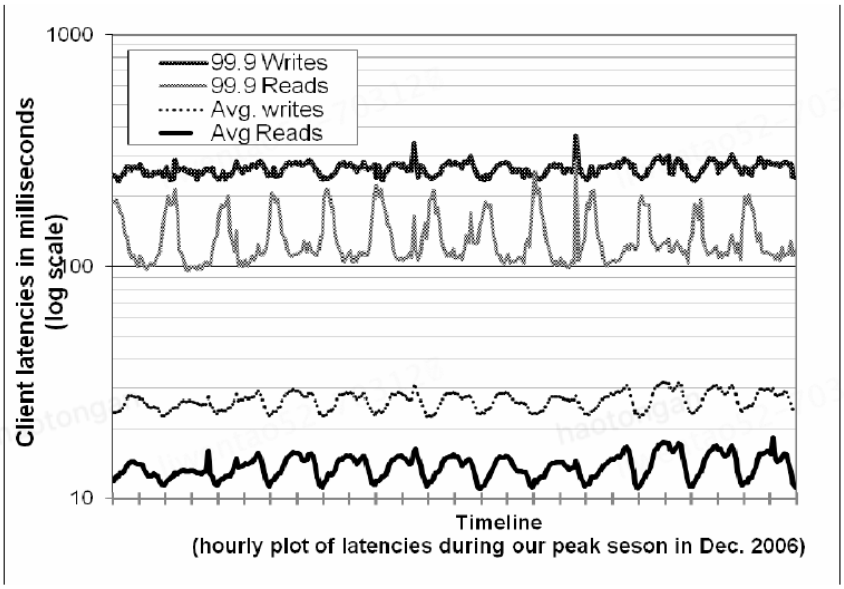


Figure 4: Average and 99.9 percentiles of latencies for read and write requests during our peak request season of December 2006. The intervals between consecutive ticks in the x-axis correspond to 12 hours. Latencies follow a diurnal pattern similar to the request rate and 99.9 percentile latencies are an order of magnitude higher than averages

上图显示了30天内 Dynamo 的读操作和写操作的平均值和P99.9的延迟。 延迟呈现出明显的日变化模式（diurnal pattern），白天和夜间存在明显的差异，这和每天请求频率的趋势是一致的。此外，写延迟明显要高于读延迟，因为写操作总是会访问磁盘。其P99.9延迟约为200ms，比平均值高一个数量级。这是因为P99.9受到多个因素的影响，例如请求负载的变化、以及对象大小和位置模式等。

虽然这种性能级别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的性能。 对于这些服务，Dynamo 提供了在持久性保证和性能之间进行权衡的能力。 在优化中，每个存储节点在其主内存中维护一个对象缓冲区。 每个写操作都写入缓冲区中，并由独立的写入线程定期写入磁盘中。 在此方案中，读操作首先检查请求的 key 是否存在于缓冲区中。 如果是，则从缓冲区中获取对象而不是从存储引擎来读取对象。

这种优化使得在高峰流量期间将P99.9的延迟降低了5倍，即使对于只有1000个对象的非常小的缓冲区效果也很明显。

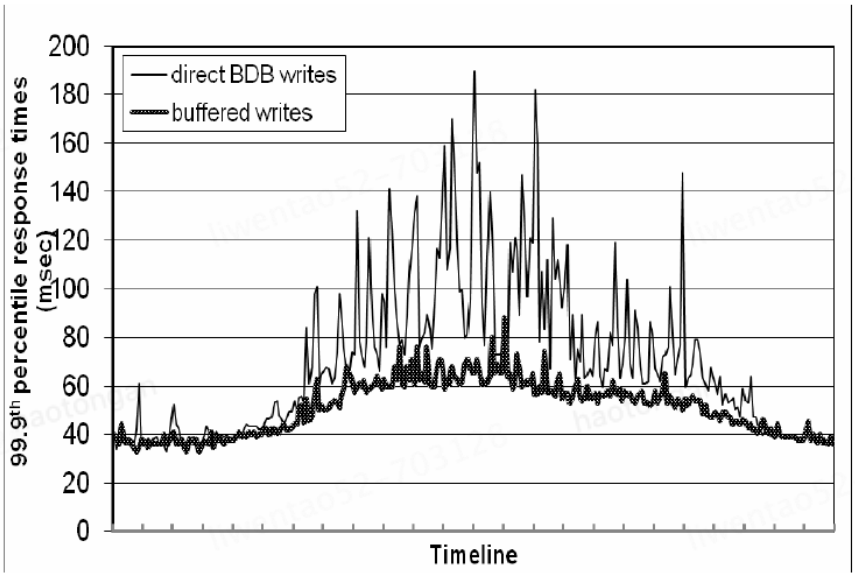


Figure 5: Comparison of performance of 99.9th percentile latencies for buffered vs. non-buffered writes over a period of 24 hours. The intervals between consecutive ticks in the x-axis correspond to one hour.

写缓冲可以降低较高的百分位数延迟。 显然，该方案牺牲了持久性来换取性能上的提升。 在此方案中，服务器崩溃可能会导致缓冲区中排队的写入丢失。 为了避免这种风险，对写操作进行了细化，coordinator 会从 N 个副本中选择一个来执行“持久化写”，也就是写入存储引擎中。 由于 coordinator 仅等待 W 个节点响应，就会返回写成功，因此写操作的性能不会受到单个副本执行的持久化写入的性能影响（持久化写入的节点响应可能会比较慢，但当其余写缓冲区节点响应完成后，就写成功了）。

6.2 确保均匀负载分布

Dynamo 使用一致性哈希对 key 空间跨多副本进行分区，确保负载分布的均衡。假设 key 的访问分布不是高度倾斜的，统一的 key 的分布可以帮助我们实现均匀的负载。特别的，即使在访问分布中存在显著的倾斜，只要活跃的 key 的数量足够多，那么就可以通过分区（partitioning）在多个节点之间均匀地分配处理这些活跃的 key 的负载。也就是说，它的设计旨在确保系统能够通过在节点间分散存储和处理请求，来均匀处理高负载的活跃的 key，从而避免某个单一节点因处理过数据而成为系统瓶颈。

本节讨论 Dynamo 中出现的负载不平衡以及不同分区策略对负载分配的影响。

为了研究负载不平衡及其与请求负载的相关性，测量了24小时内每个节点收到的请求总数，每30分钟作为一个时间窗口。 在给定的时间窗口内，如果节点的请求负载与平均负载的偏差小于某个阈值（此处为 15%），则该节点被视为“平衡”。否则该节点被视为“失衡”。

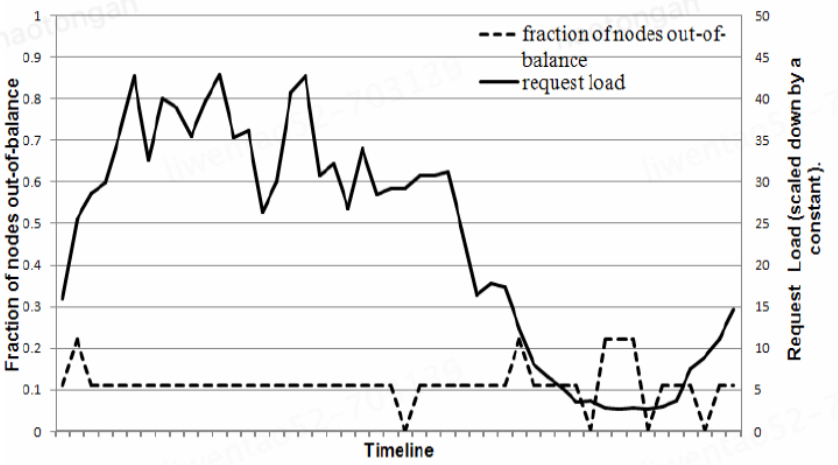


Figure 6: Fraction of nodes that are out-of-balance (i.e., nodes whose request load is above a certain threshold from the average system load) and their corresponding request load. The interval between ticks in x-axis corresponds to a time period of 30 minutes.

上图显示了这段时间内“失衡”节点的比例（虚线），作为参考，还绘制了整个系统在此时间段内接收到的相应请求负载（实线），可以观察到“失衡”节点的比例随着请求负载的增加而减小。例如，在低负载期间，失衡比例高达 20%，在高负载期间，不平衡率接近 10%。 直观上，这可以通过以下事实来解释：在高负载下，会访问大量活跃的 key，请求会均匀分发到各个节点上，整个系统负载均衡。低负载下（峰值1/8），活跃 key 访问的数量较少，会导致负载不均衡。

本节讨论 Dynamo的分区方案是如何随着时间的推移而演变的，以及它对负载分配的影响。

Strategy 1: T random tokens per node and partition by token value

这是在生产环境中部署的初始策略（4.2节）。在这个方案中，每个节点被分配 T 个令牌（从哈希空间中随机均匀选择）。所有节点的令牌按照它们在哈希空间中的值排序。每两个连续的令牌定义一个范围。 最后一个令牌和第一个令牌相连，形成一个环空间。由于令牌是随机选择的，因此范围大小各不相同。 当节点加入和离开系统时，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的。

使用中出现的问题，首先，当一个新节点加入系统时，它需要从其他节点窃取（steal”）它应负责的 key 范围。但是，将 key 范围传递给新节点的节点必须扫描其本地持久化存储以检索适当的数据项集。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（磁盘I/O）的操作，并且它们需要在后台执行，而不影响性能。这就要求我们以最低优先级运行该任务。 然而，这显着减慢了节点上线的过程，并且在繁忙的购物季节，当节点每天处理数百万个请求时，几乎需要一天才能完成节点上线。

其次，当节点加入/离开系统时，许多节点处理的键范围发生变化，并且需要重新计算新范围的 Merkle tree，这是在生产系统上执行同样是一项重要操作。

最后，由于 key 范围的随机性，没有简单的方法来获取整个 key 空间的快照，这使得归档过程变得复杂。 在该方案中，归档整个 key 空间需要我们分别从每个节点检索 key，效率非常低。

该策略的根本问题是数据分区和数据放置的方案是相耦合的。 例如，在某些情况下，最好向系统添加更多节点，以便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。

Strategy 2: T random tokens per node and equal sized partitions

该策略将哈希空间被划分为 Q 个大小相等的分区/范围（partition/range），每个节点分配 T 个随机令牌。Q 通常设置的值要远大于 N 以及，远大于 S*T（Q >> N和Q >> S*T），其中 S 为系统节点数。在此策略中，令牌仅用于构建将哈希空间中的值映射到有序节点列表的函数，而不用于决定数据分区。分区数据被放置在从分区末尾顺时针遍历一致性哈希环时遇到的前 N 个唯一节点上。

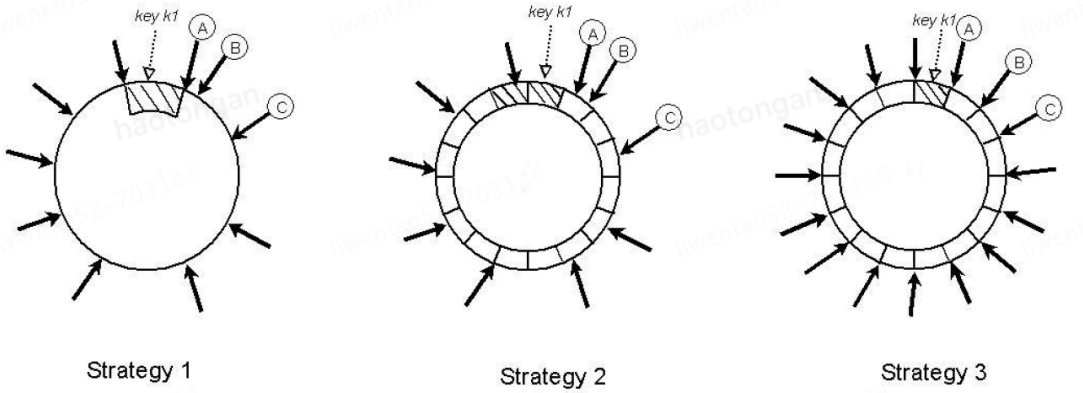


Figure 7: Partitioning and placement of keys in the three strategies. A, B, and C depict the three unique nodes that form the preference list for the key k1 on the consistent hashing ring (N=3). The shaded area indicates the key range for which nodes A, B, and C form the preference list. Dark arrows indicate the token locations for various nodes.

在这个示例中，从包含 key k1 的分区的末端遍历环时，会遇到节点A、B、C。该策略的主要优点是

- 分区和放置解耦
- 允许在运行时更改放置方案

Strategy 3: Q/S tokens per node, equal-sized partitions

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 是系统中节点的数量。 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，以便保留这些性质（Q/S tokens per node）。 类似地，当节点加入系统时，它会以保留这些性质的方式从系统中的节点“窃取”令牌。这种方法在一定程度上确保了每个节点对存储和管理数据负有平等的责任。

使用 S=30 和 N=3 的系统，评估了这三种策略的效率。当然，公平比较这三种策略是困难的，策略1的 负载分布属性取决于令牌的数量（T），而策略3取决于分区的数量（Q）。较公平的方式采用了，所有的策略都使用相同大小的空间存储成员资格信息时，测量它们的负载分布倾斜度。例如，策略1中每个节点需要为环上的全部节点维护各自的令牌的位置，而策略3中每个节点需要维护系统分配给每个节点的分区信息。

通过改变相关参数（T和Q）来评估这些策略。针对每个节点需要维护的成员资格信息的不同大小，测量每种策略的负载均衡效率，其中负载均衡效率（load balancing efficiency）定义为每个节点平均请求数与负载最高的节点的最大请求数的比值。

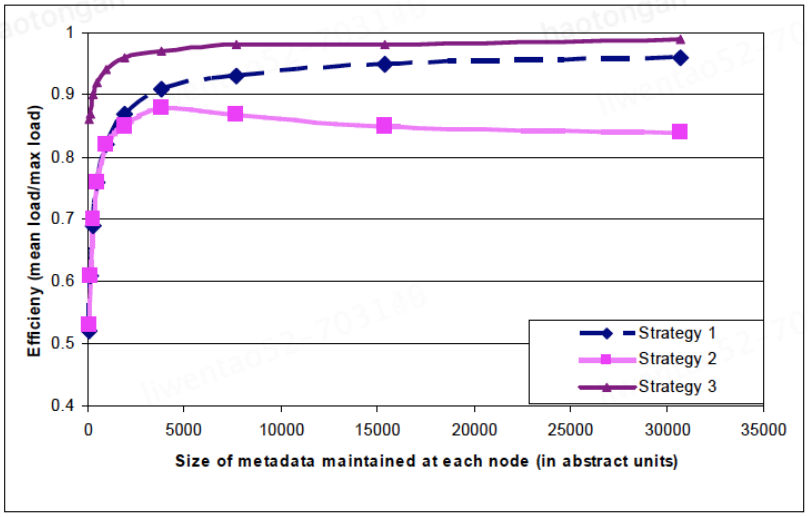


Figure 8: Comparison of the load distribution efficiency of different strategies for system with 30 nodes and N=3 with equal amount of metadata maintained at each node. The values of the system size and number of replicas are based on the typical configuration deployed for majority of our services.

策略3的负载均衡效率最好，策略2的负载均衡效率最差。从使用策略1迁移到策略3的过程中，策略2曾短暂地用作临时设置过度策略。与策略1相比，策略3的效率更高，并且减少了每个节点所需维护的成员信息。虽然存储这些信息不是主要问题，但是节点之间会周期性的交换成员信息（gossip），所以还是要尽可能的保持这些信息的紧凑性。此外，策略3的部署更容易：

Faster bootstrapping/recovery

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，这意味着只需传输文件即可将分区作为一个单元来重新放置（relocated）到新节点处（增加/删除节点时，可以将对应的分区进行传输，避免定位特定项目所需的随机访问）。 这简化了 bootstrapping 和 recovery 的过程。

Ease of archival

定期归档数据集是大多数 Amazon 存储服务的强制性要求。在策略3中，归档 Dynamo 存储的整个数据集更简单，因为分区文件可以单独归档。

在策略1中，由于令牌是随机选择的，存储的数据归档时需要分别从各个节点检索key，通常效率低下且缓慢。 策略3当节点加入（扩容）或离开（缩容）集群时，为了保持其属性（Q/S tokens per node），需要进行一定的协调工作。

6.3 分支版本: 何时？有多少?

上文提及了 Dynamo 被设计为在一致性和可用性间权衡（tradeoff consistency for availability），要了解不同故障对一致性的确切影响，需要多个因素的详细数据，包括停机时间、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。然而，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。

存在不同版本的数据会在两种情况下出现。 第一种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。 第二种情况是当系统正在处理单个数据项的大量并发写入，并且多个节点最终同时协调更新时。 从可用性（usability）和效率的角度来看，最好在任何时间都保持不同版本的数据的数量尽可能低。 如果无法根据向量时钟进行句法协调，则必须将它们返回给客户端，在业务逻辑上进行语义协调。 语义协调会给服务带来额外的负担，因此最好减少。

对返回到购物车服务的版本数进行了24小时的分析。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。这表明很少产生不同的版本。

经验表明，不同版本数量的增加通常不是由故障造成的，而是由于并发写入数量的增加造成的。 并发写入数量的增加通常是由繁忙的机器人（自动化客户端程序）触发的，很少人为触发的。 由于案例的敏感性，这个问题不进行详细讨论。

6.4 客户端驱动or服务端驱动协调

第5节提到，Dynamo 有一个请求协调组件，它使用状态机来处理传入请求。 客户端请求由负载均衡器统一分配到环上的节点。 任何 Dynamo 节点都可以充当读取请求的coordinator。 另一方面，写入请求将由 key当前偏好列表中的节点作为 coordinator 来协调写入。 这是由于偏好列表中的节点具有创建新版本标记的额外责任，写入请求更新的版本就自然而然的由这些节点来标记上。 注意，如果 Dynamo 的版本控制方案基于物理时间戳，则任何节点都可以作为 coordinator 来协调写入请求。

考虑由客户端驱动协调写入，客户端应用使用库（library）在本地执行请求协调。客户端定期选择一个随机的 Dynamo 节点并下载其当前的 Dynamo 系统成员状态视图。使用此信息，客户端就可以确定任意 key 的偏好列表中的节点集合。读请求可以在客户端节点协调，从而避免了负载均衡器将请求分配给随机的 Dynamo节点时产生的额外网络转发跳数（hop）。写操作要么转发给 key 对应的偏好列表里面的一个节点，那么如果使用的是基于物理时间戳的版本化方式 ，可以在本地协调写入。

客户端驱动的一个重要优点是：不再需要负载均衡器统一分配客户端的请求负载。 在存储节点上近乎均匀分布的 key，也隐约保障了负载的均匀分布（一致性哈希，使得系统节点的负载已经是均匀的了）。显然，该方案的效率取决于系统成员信息在客户端的新旧程度。目前客户端每10秒轮询一个随机的 Dynamo 节点来获取成员信息的更新。这里选择了基于拉的方法，而不是基于的推的方法，因为前者更适合大量的客户端，并且只需要在服务器上维护较少的关于客户端的状态信息。然而，在最坏的情况下，客户端可能会在10秒的时间内持有过时的状态信息。 如果客户端检测到其成员资格表已过时（例如，当某些成员无法访问时），它将立即更新其成员资格信息。

Table 2: Performance of client-driven and server-driven coordination approaches.

	99.9th percentile read latency (ms)	99.9th percentile write latency (ms)	Average read latency (ms)	Average write latency (ms)
Server-driven	68.9	68.5	3.9	4.02
Client-driven	30.4	30.4	1.55	1.9

不难发现，客户端驱动的方式比服务端方式P99.9减少了至少30ms，平均延迟减少了3～4ms。延迟降低的主要原因就是客户端驱动的方式避免了负载均衡器的开销，减少了网络跳数，延迟自然就降低了。平均延迟 显著低于P99.9，是因为 Dynamo 的存储引擎缓存和写缓冲区有很好的命中率。此外，由于负载均衡器和网络会给延迟引入额外的可变因素（variability），因此P99.9的性能提升要比平均性能提升更明显。

6.5 平衡后台与前台任务

除了执行正常的前台读/写操作外，每个节点还执行不同类型的后台任务，用于副本同步和数据切换（hinting or adding/removing nodes）。在早期的生产环境中，这些后台任务触发了资源竞争问题，并影响了 常规读/写操作的性能。因此，有必要确保后台任务只在正常关键操作不受显著影响的情况下运行。为此，将后台任务与流量控制机制（admission control mechanism）集成在一起。每个后台任务都使用这个控制器来保留资源(例如数据库)的运行时时间片（reserve runtime slices），这些资源是在所有后台任务之间共享的。基于前台任务性能的监控会通过反馈机制来改变后台任务可以使用的时间片的数量。

在执行读/写操作时，控制器将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、 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等。这些信息用于检查在给定的跟踪时间窗口中延迟(或失败)的性能是否接近所需的阈值。控制器检查数据库（60s）的读延迟P99性能是否与预设的阈值（50ms）足够近。控制器使用这种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，从而使用反馈回路（feedback loop）来限制后台任务。

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。许多 Amazon 内部服务在过去两年中都在使用 Dynamo，并且它为应用程序提供了相当高的可用性。特别地，应用程序有99.9995%请求的收到了成功响应（不包括超时），并且到目前为止没有发生数据丢失的事件。

- Dynamo 提供了必要的配置能力，调整N,R,W参数来权衡持久性和可用性。
- 同时，也数据一致性和协调逻辑开放给开发人员（应用程序设计就应考虑各种故障情况和数据不一致）。
- 采用完全成员关系模型（full membership model），其中每个节点都知道其对等节点存储了哪些数据（几百上千个节点通过 gossip 机制完全没问题，但上万个就有些困难了，需要引入新的机制来协调，例如DHT systems）。

7.结论

本文描述了 Dynamo，一个高度可用且可扩展的数据存储，用于存储 Amazon 电子商务平台的许多核心服务的状态。 Dynamo 提供了满足需要的可用性和性能等级，并正确处理服务器故障、数据中心故障和网络分区等。 Dynamo 具有增量可扩展性，允许服务所有者根据当前请求负载进行扩容和缩减。 Dynamo 允许服务所有者调整参数 N、R 和 W，从而定制其存储系统，以满足他们所需的性能、持久性和一致性 SLA 需求。

过去一年 Dynamo 的生产使用表明，可以组合分散的技术来提供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的成功表明，最终的一致性存储系统可以成为高可用应用程序的基础。

参考

略

相关文章

1.Dynamo: Amazon’s Highly Available Key-value Store

2.Amazon DynamoDB: A Scalable, Predictably Performant, and Fully Managed NoSQL Database Service

3.https://zhuanlan.zhihu.com/p/342178343

4.https://en.wikipedia.org/wiki/Consistent_hashing

5.https://queue.acm.org/detail.cfm?id=2917756

6.https://en.wikipedia.org/wiki/Vector_clock

7.https://en.wikipedia.org/wiki/Merkle_tree

8.https://zhuanlan.zhihu.com/p/556601917
9.Designing Data-Intensive Applications