刊登于*第 23 届 ACM 操作系统原理研讨会*

(SOSP'11) 论文集

不要满足于最终结果

利用 COPS 实现广域存储的可扩展因果一致性

Wyatt Lloydx、Michael J. Freedmanx、Michael Kaminsky†和 David G. Andersen‡

x 普林斯顿大学、† 英特尔实验室、‡ 卡内基梅隆大学

摘要

支持复杂在线应用（如社交网络）的地理复制分布式数据存储必须提供 "始终在线 "的体验，即始终以较低的延迟完成操作。当今的系统为了实现这些目标，往往牺牲了较强的一致性，从而将不一致性暴露给客户端，并需要复杂的应用逻辑。在本文中，我们确定并定义了一种一致性模型--具有收敛冲突处理功能的因果一致性，或称*因果+*，它是在这些限制条件下实现的最强一致性。

我们介绍了 COPS 的设计与实现，它是一种能在广域范围内提供这种一致性模型的键值存储。COPS 的主要贡献在于其可扩展性，它可以在整个集群而不是像以前的系统那样在单个服务器上执行键值存储之间的因果依赖关系。COPS 的核心方法是跟踪并明确检查密钥之间的因果依赖关系是否在本地集群中得到满足，然后再进行写入。此外，在 COPS-GT 中，我们还引入了获取事务（get transactions），以便在不锁定或阻塞的情况下获得多个密钥的一致视图。我们的评估结果表明，COPS 能在不到一毫秒的时间内完成操作，在每个集群使用一台服务器时，其吞吐量与以前的系统相似，而且随着每个集群服务器数量的增加，其吞吐量也在不断增加。评估还表明，对于常见工作负载，COPS-GT 可提供与 COPS 类似的延迟、吞吐量和扩展能力。

类别和主题描述符

C.2.4 [**计算机系统组织**]：分布式系统

一般条款

设计、实验、表演

关键词

键值存储、因果+一致性、可扩展广域复制、ALPS 系统、读取事务

允许将本作品的全部或部分内容制作成数字或硬拷贝，供个人或课堂使用，不收取任何费用，但不得以营利或商业利益为目的制作或分发拷贝，且拷贝必须在首页上标明本声明和完整的引文。如需复制、再版、在服务器上发布或在列表中重新分发，则需事先获得特别许可和/或付费。

SOSP '11，2011 年 10 月 23-26 日，葡萄牙卡斯卡伊斯。

版权 2011 ACM 978-1-4503-0977-6/11/10 .. . $10.00.

1. 引言

分布式数据存储是现代互联网服务的基本构件。在理想情况下，这些数据存储应具有很强的一致性，随时可供读写，并能在网络分区时继续运行。遗憾的是，CAP 定理证明不可能创建一个同时实现这三点的系统 [13, 23]。相反，现代网络服务以牺牲强一致性为代价，过度追求可用性和分区容忍性 [16, 20, 30]。这也许并不令人惊讶，因为这种选择还能使这些系统为客户端操作提供低延迟和高可扩展性。此外，许多早期的大规模互联网服务（通常侧重于网络搜索）认为没有什么理由需要更强的一致性，不过随着社交网络应用程序等交互式服务的兴起，这种观点正在发生变化[46]。我们将具有这四个**特性**--可用性、低延迟、分区容忍性和高可扩展性--的系统称为 ALPS 系统。

鉴于 ALPS 系统必须牺牲强一致性（即线性化），我们寻求在这些限制条件下可实现的最强一致性模型。较强的一致性是可取的，因为它能使程序员更容易对系统进行推理。在本文中，我们考虑的是*具有收敛冲突处理功能的因果一致性*，我们称之为*因果+ 一致性*。以前的许多系统被认为实现了较弱的因果一致性[10, 41]，但实际上实现了更有用的因果+一致性，不过没有一个系统是以可扩展的方式实现的。

causal+ 一致性的因果组件确保数据存储尊重操作之间的因果依赖关系[31]。考虑这样一种情况：用户向网站上传图片，图片被保存，然后该图片的引用被添加到该用户的相册中。该引用 "依赖于 "被保存的图片。在因果+ 一致性条件下，这些依赖关系总是得到满足。程序员永远不必像在最终一致性等保证较弱的系统中那样，遇到能获得图片引用而不能获得图片本身的情况。

因果+一致性的收敛冲突处理组件可确保复制永远不会出现分歧，而且所有站点都会以相同方式处理同一密钥的冲突更新。当与因果一致性相结合时，这一特性确保客户端只能看到密钥的逐步更新版本。相比之下，最终一致性系统可能会暴露出不按顺序排列的版本。通过将因果一致性和收敛冲突处理相结合，因果+ 一致性可确保客户端看到因果正确、无冲突且始终在进步的数据存储。

我们的 COPS 系统（Clusters of Order-Preserving Servers，保序服务器集群）提供因果+一致性，旨在支持复杂的在线应用程序，这些应用程序由少数大型数据中心托管，每个数据中心都由前端服务器（Client of OrderPreserving Servers，保序服务器集群）和后端服务器（Client of OrderPreserving Servers，保序服务器集群）组成。

COPS）和后端键值数据存储。COPS 在本地数据中心以可线性化的方式执行所有放入和获取操作，然后在后台以因果+一致的顺序跨数据中心复制数据。

我们详细介绍了 COPS 系统的两个版本。常规版本 COPS 在数据存储中的单个项目之间提供可扩展的因果+一致性，即使它们的因果依赖关系分布在本地数据中心的许多不同机器上。这些一致性特性的成本很低：COPS 的性能和开销与之前的系统（如基于日志交换的系统 [10，41]）相似，但可扩展性更高。

我们还详细介绍了该系统的扩展版本 COPS-GT，它也提供*获取事务*，为客户提供多个密钥的一致视图。即使在完全线性化的系统中，也需要获取事务来获得多个密钥的一致视图。我们的获取事务不需要锁，是非阻塞的，最多需要两轮并行的数据中心内请求。据我们所知，COPS-GT 是首个实现非阻塞可扩展获取事务的 ALPS 系统。但这些事务也付出了一些代价：与普通版本的 COPS 相比，COPS-GT 在某些工作负载（如写入量大）上的效率较低，对长网络分区和数据中心故障的鲁棒性也较差。

ALPS 系统的可扩展性要求是 COPS 与之前的因果+一致系统之间最大的区别。以前的系统要求所有数据都必须放在一台机器上[2, 12, 41]，或者所有可能被一起访问的数据都必须放在一台机器上[10]。相比之下，COPS 中存储的数据可以分布在任意大小的数据中心，依赖关系（或获取事务）可以延伸到数据中心的许多服务器上。据我们所知，COPS 是第一个实现因果+（也就是因果）一致性的可扩展系统。

本文的贡献包括

- 我们明确指出了分布式数据存储的四个重要属性，并用它们来定义 ALPS 系统。
- 我们命名并正式定义了因果+一致性。
- 我们介绍了 COPS 的设计和实施，这是一个*可扩展的*能有效实现因果+ 一致性模型的系统。
- 我们在 COPS-GT 中提出了一种无阻塞、无锁的获取事务算法，它能在最多两轮本地操作中为客户提供多个密钥的一致视图。
- 通过评估，我们发现 COPS 延迟低、吞吐量高，而且在所有测试的工作负载中都能很好地扩展；对于常见的工作负载，COPS-GT 也具有类似的特性。

2. 阿尔卑斯系统和权衡

我们感兴趣的基础设施可以支持当今许多最大型的互联网服务。传统的分布式存储系统侧重于小范围内的局域操作，与之相比，这些服务的典型特点是跨几个到几十个数据中心的广域部署，如图 1 所示。每个数据中心都包括一组应用级客户机，以及这些客户机读写的后端数据存储。对于许多应用--也是本文所考虑的应用--来说，在一个数据中心写入的数据会复制到其他数据中心。

通常情况下，这些客户端实际上是代表远程浏览器运行代码的网络服务器。虽然本文是从应用客户端（即网络服务器）的角度来考虑一致性的，但如果浏览器通过单一数据中心访问服务，正如我们所期望的那样，它也将享有类似的一致性保证。

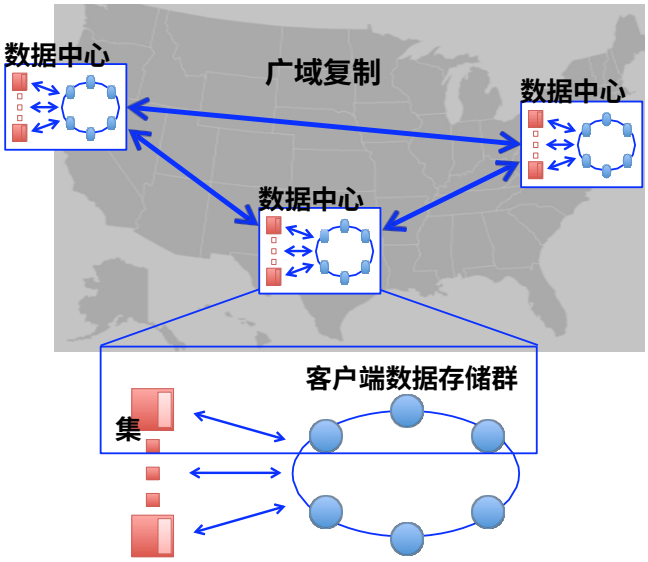


图 1：现代网络服务的总体架构。多个分布在不同地理位置的数据中心都有应用客户端，客户端从分布在所有数据中心的数据存储中读取和写入状态。

这种分布式存储系统有多重目标，有时甚至是相互矛盾的目标：*可用性*、*低延迟*和*分区容错*，以提供 "始终在线 "的用户体验[16]；*可扩展性*，以适 应 不断增加的负载和存储需求；足够强大的*一致性*模型，以简化编程并为用户提供他们所期望的系统行为。更深入地讲，理想的特性包括

1. 可用性。对数据存储发出的所有操作都能成功完成。任何操作都不能无限期阻塞或返回错误，表示数据不可用。

2. 低延迟。客户端操作 "快速 "完成。商业服务级目标表明，平均性能为几毫秒，最差情况下（即 99.9 百分位数）为 10 或 100 毫秒[16]。

3. 分区容错。数据存储继续在网络分区下运行，例如，将亚洲数据中心与美国数据中心分隔开来。

4. 高扩展性。数据存储可线性扩展。向系统添加 *N* 个资源后，总吞吐量和存储容量将增加 O(*N*)。

5. 更强的一致性。理想的数据存储应提供*可线性化（有时非正式地称为强一致性），即*在调用和完成操作之间的单个时间点上，操作似乎在整个系统中生效[26]。在提供线性化的数据存储中，只要客户端在一个数据中心完成了对对象的写操作，在所有其他数据中心对同一对象的读操作都将反映其新写入的状态。线性化简化了编程--分布式系统提供了单一、一致的映像，用户体验到了他们所期望的存储行为。在许多大型分布式系统中常见的较弱的最终一致性模型就不那么直观了：不仅后继读取可能无法反映最新值，跨多个对象的读取也可能反映新旧值的不连贯混合。

CAP 定理证明，具有可用性和分区容差的共享数据系统无法实现线性化[13]、

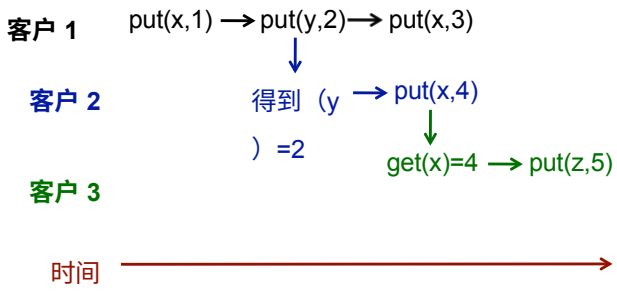


图 2：图示操作和维护之间的因果关系。在一个副本中的关系。从 *a* 到 *b* 的边表示 *a* ~ *b*，或 *b* 取决于 *a*。

23].低延迟定义为小于副本之间最大宽域延迟的延迟，这也被证明与线性化[34]和顺序一致性[8]不相容。为了平衡 ALPS 系统的要求和可编程性，我们在下一节定义了一个中间一致性模型。

3. 因果+一致性

要定义*具有收敛冲突处理功能的因果一致性*（因果+ 一致性），我们首先要描述其运行的抽象模型。我们只考虑键值数据存储，其中有两个基本操作：`put`(key,val) 和 `get`(key)=val。这相当于共享内存系统中的写和读操作。值从逻辑*副本中*存储和检索，每个*副本*都承载着整个键值空间。在我们的 COPS 系统中，单个*逻辑副本*对应整个本地节点集群。

我们模型中的一个重要概念是操作之间的*潜在*因果关系[2, 31]。有三条规则定义了潜在因果关系，用 ~ 表示：

1. **执行线程。**如果 *a* 和 *b* 是*单个执行线程*中的两个操作，那么如果操作 *a* 发生在操作 *b* 之前，则 *a* ~ *b*。
2. **从获取。**如果 *a* 是一个 `put` 操作，*b* 是一个 `get` 操作，返回 *a* 写入的值，那么 *a* ~ *b*。
3. **传递性。**对于运算 *a*、*b* 和 *c*，如果 *a* ~ *b* 和 *b* ~ *c*，那么 *a* ~ *c*。

这些规则在同一执行线程内的操作之间以及执行线程通过数据存储进行交互的操作之间建立了潜在的因果关系。与许多模型一样，我们的模型不允许线程直接通信，而是要求所有通信都通过数据存储进行。

图 2 中的执行示例演示了这三种规则。*执行线程*规则给出了 `get`(*y*)=2 ~ `put`(*x*,4)；*从*规则给出了 `put`(*y*,2) ~ `get`(*y*)=2；*反向*规则给出了 `put` (*y*,2) ~ `put`(*x*,4)。尽管有些操作实时跟随 `put`(*x*,3)，但没有其他操作依赖于它，因为例如，在图 2 中，`put`(*y*,2) 似乎发生在 `put`(*x*,4)，而这又发生在 `put`(*z*,5) 之前。如果客户 2 看到得到（*x*）=4，然后得到（*x*）=1，这就违反了因果一致性。因果一致性不对并发操作排序。如果 *a* /~ *b* 和 *b* /~ *a*，那么 *a* 和 *b* 是并发的。通常，这可以提高执行效率：两个互不相关的操作它们可以以任何顺序复制，从而避免了在它们之间设置序列化点。但是，如果 *a* 和 *b* 都指向同一个键，那么它们就会*发生冲突*。

没有任何操作读取它写入的值，也没有任何操作在同一执行线程中跟随它。

3.1 定义

我们将因果+一致性定义为两个属性的组合：因果一致性和收敛冲突处理。我们在此给出直观定义，正式定义见附录 A。*因果一致性*要求从副本的获取操作返回的值与 ~ 定义的顺序一致（因果关系）[2]。换句话说，写入值的操作必须出现在所有因果关系在前的操作之后。

冲突之所以不可取，有两个原因。首先，由于冲突不按因果一致性排序，因此冲突会让复制永远背离[2]。例如，如果 *a* 是 put(*x*,1)，而 *b* 是 put(*x*,2)，那么因果一致性允许一个副本永远为 *x* 返回 1，而另一个副本永远为 *x* 返回 2。例如，在购物车应用程序中，如果两个登录到同一账户的人同时向购物车中添加物品，那么理想的结果是两个物品都出现在购物车中。

*趋同冲突处理*要求在所有副本中以相同的方式处理所有冲突的提交，使用处理程序函数
*h.*处理函数 *h* 必须具有关联性和交换性，这样各副本就能按照接收到的顺序处理冲突的写入，并且这些处理结果会趋同（例如，一个副本的 *h*(*a*,*h*(*b*,*c*))和另一个副本的 *h*(*c*,*h*(*b*,*a*))一致）。

以趋同方式处理冲突写入的一种常见方法是 "最后写入者获胜 "规则（也称托马斯写入规则 [50]），该规则将冲突写入中的一个写入声明为发生在后，并覆盖 "较早 "的写入。处理冲突写入的另一种常见方法是将它们标记为冲突写入，并要求通过其他方式解决，例如通过 Coda [28] 中的直接用户干预，或通过 Bayou [41] 和 Dynamo [16] 中的编程程序。

所有潜在的聚合冲突处理形式都能避免第一个问题--冲突更新可能会持续发散--确保副本在交换操作后得到相同的结果。另一方面，冲突的第二个问题--应用程序可能希望对冲突进行特殊处理--只有通过使用更明确的冲突*解决*程序才能避免。这些显式程序为应用程序提供了更大的灵活性，但需要额外的程序复杂度和/或性能开销。虽然 COPS 可以配置为自动检测冲突更新，并应用一些应用程序定义的解决方法，但 COPS 的默认版本使用的是 "最后写入者获胜 "规则。

3.2 因果+与其他一致性模型的比较

分布式系统文献定义了几种流行的一致性模型。按强度递减，它们包括线性化（或强一致性）[26]，保持全局实时排序；顺序一致性[32]，至少确保全局排序；因果一致性[2]，确保依赖操作之间的部分排序；先进先出（PRAM）一致性[34]，它只保留执行线程的部分排序，而不保留线程之间的排序；按键顺序一致性[15]，它确保对于每个单独的键，所有操作都有一个全局顺序；以及最终一致性，这是一个 "包罗万象 "的术语，表示最终收敛到某种类型的协议。

如图 3 所示，我们引入的因果+ 一致性介于顺序一致性和因果一致性之间。它比顺序一致性弱，但顺序一致性在 ALPS 系统中是无法实现的。它比因果一致性更强

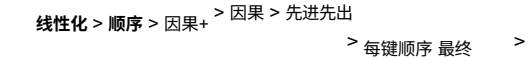


图 3：一致性模型频谱，左侧为较强的模型。加粗的模型与 ALPS 系统不兼容。

然而，ALPS 系统可以实现因果一致性和按键顺序一致性。Mahajan 等人[35] 同时定义了类似的因果一致性强化；详情请参见第 7 节。为了说明因果+模型的实用性，请看两个例子。

首先，让爱丽丝尝试与鲍勃分享一张照片。爱丽丝上传照片，然后将照片添加到她的相册中。然后鲍勃查看爱丽丝的相册，希望看到她的照片。在因果一致性和因果+ 一致性条件下，如果相册中有对照片的引用，那么鲍勃一定能看到照片。

在按键顺序一致性和最终一致性下，相册有可能有一个引用的照片。

其次，举个例子，卡罗尔和丹同时更新了一个事件的开始时间。时间最初设置为晚上 9 点，卡罗尔将其改为晚上 8 点，而丹同时将其改为晚上 10 点。常规的因果一致性会允许两个不同的副本返回不同的时间，即使同时接收了两个放操作。因果+一致性要求副本以趋同的方式处理这种冲突。如果使用 "最后写入者获胜 "策略，那么无论是 Dan 的 10pm 还是 Carol 的 8pm 都会获胜。如果使用更明确的冲突解决策略，则可将该密钥标记为冲突密钥，将来对该密钥的获取可同时返回 8pm 和 10pm，并说明如何解决冲突。

如果数据存储是顺序一致的或可线性化的，那么仍有可能同时对一个密钥进行两次更新。不过，在这些更强的模型中，可以实现互斥算法，比如 Lamport 在最初的顺序一致性论文 [32] 中提出的互斥算法，这种算法可以完全避免产生冲突。

3.3 COPS 中的 Causal+

我们使用 COPS 系统中的两个抽象概念--*版本*和*依赖*，来帮助我们推理因果+一致性。我们把密钥的不同值称为密钥的*版本*，并用 $_{\text{keyversion}}$ 表示。在 COPS 中，版本的分配方式是确保如果 $x_i \sim y_j$ 则 $i < j$。一旦 COPS 中的副本返回了密钥的 i 版本，即 x_i，因果+ 一致性就会确保它只会返回该版本或因果关系上更晚的版本（注意，冲突的处理在因果关系上要晚于它所解决的冲突）。[1]因此，COPS 中的每个副本总是返回密钥的非递减版本。我们把这称为因果+一致性的*渐进属性*。因果一致性规定，所有在因果关系上先于给定操作生效的操作都必须先于给定操作生效。换句话说，如果 $x_i \sim y_j$，那么 x_i 必须写在 y_j 之前。我们称这些前置值为*依赖关系*。更正式地说，当且仅当 $\texttt{put}(x_i) \sim \texttt{put}(y_j)$ 时，我们说 y_j ***依赖于*** x_i。这些依赖关系实质上是写入的因果排序的反向。COPS 在复制过程中提供因果+ 一致性，只写入一个版本写入其所有依赖项之后。

[1] 要理解这一点，请考虑以下矛盾情况：假设副本首先返回 x_i，然后返回 x_k，其中 $i \neq k$ 且 $x_i /\sim x_k$。因果一致性确保，如果 x_k 在 x_i 之后返回，则 $x_k /\sim x_i$，因此 x 在 x 之后返回。

3.4 可扩展的因果关系

据我们所知，本文是第一篇命名并正式定义因果+一致性的论文。有趣的是，之前几个被认为能实现因果一致性的系统[10, 41]实际上实现了更强的因果+一致性保证。

然而，这些系统的设计目的并不是提供*可扩展的*因果（或因果+ ）一致性，因为它们都使用一种日志序列化和交换形式。逻辑副本中的所有操作都按序列化顺序写入单个日志，通常用版本向量标记[40] 。然后，不同的副本交换这些日志，使用版本向量建立潜在的因果关系，并检测不同副本操作之间的并发性。

基于日志交换的序列化抑制了副本的可扩展性，因为它依赖于每个副本中的单个序列化点来建立排序。因此，要么密钥之间的因果关系仅限于一个节点上可存储的密钥集 [10, 15, 30, 41] ，要么单个节点（或复制状态机）必须为整个集群的所有操作提供提交排序和日志。

如下所示，COPS 通过不同的方法实现了可扩展性。每个数据中心的节点负责密钥空间的不同分区，但系统可以跟踪并执行存储在不同节点上的密钥之间的依赖关系。COPS 在与每个密钥版本相关的元数据中明确编码了依赖关系。当密钥被远程复制时，接收数据中心会在提交传入版本之前执行依赖性检查。

4. 警察系统设计

和 x_k 冲突。但是，如果 x_i 和 x_k 相冲突，那么收敛冲突处理
这就确保了只要这两者都出现在副本中，它们的处理结果 $h(x_i, x_k)$ 就会返回，而不是 x_i 或 x_k ，这与我们的假设相矛盾。

COPS 是一种分布式存储系统，它实现了因果+ 一致性，并具有所需的 ALPS 特性。该系统有两个不同的版本。第一个版本我们简称为 COPS ，它提供了一个因果+ 一致性的数据存储。第二个版本称为 COPS-GT ，它提供了这一功能的超集，同时还引入了对 *get 事务的*支持。在获取事务中，客户端请求一组键，数据存储会回复相应值的一致性快照。由于强制执行 get 事务的一致性属性需要额外的元数据，因此特定部署必须完全以 COPS 或 COPS-GT 的形式运行。

4.1 COPS 概览

COPS 是一种键值存储系统，设计用于在少量数据中心之间运行，如图 4 所示。每个数据中心都有一个本地 COPS *集群*，其中包含存储数据的完整副本。[2]COPS *客户端*是使用 COPS *客户端库*直接调用 COPS 键值存储的应用程序。客户端只与其在同一数据中心运行的本地 COPS 集群通信。

每个本地 COPS 集群都被设置为可线性化（强一致性）键值存储[5, 48]。通过将密钥空间划分为 N *个*可线性化分区（每个分区可位于单个节点或单个节点链上），并让客户端独立访问每个分区，可线性化系统就能以可扩展的方式实施。可线性化的可组合性 [26] 确保所产生的系统作为一个整体保持可线性化。可线性化在本地是可以接受的，因为我们期望的是极低的延迟和无延迟。

[2] 完全复制的假设简化了我们的表述，不过我们也可以想象，集群只复制全部数据存储的一部分，其余部分则牺牲低延迟（根据配置规则）。

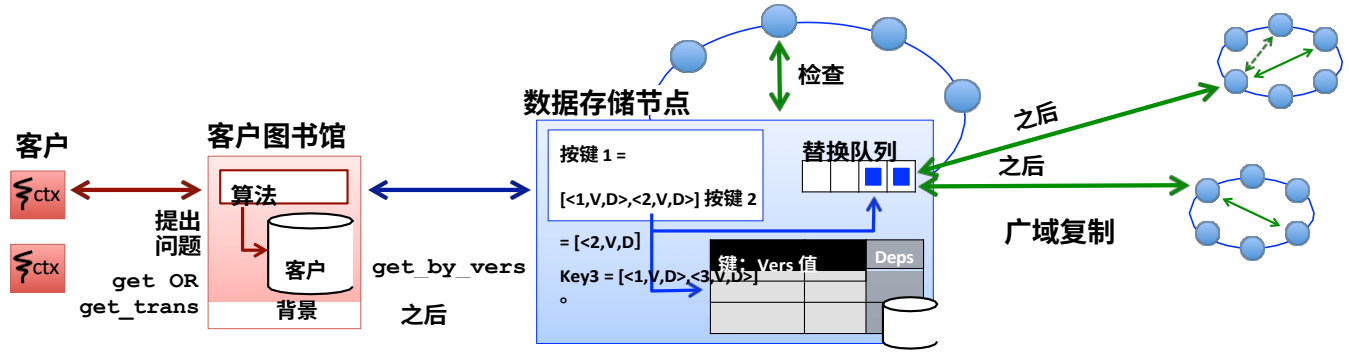


图 4：COPS 架构。客户端库向其客户端公开一个 put/get 接口，并确保操作正确标注因果依赖关系。键值存储可在集群间复制数据，确保只有在依赖关系得到满足后，才会在本地集群中提交写入，并在 COPS-GT 中存储每个键的多个版本以及依赖关系元数据。

与广域复制不同的是，COPS 的复制需要在集群内的分区中进行，特别是在现代数据中心网络的冗余路径趋势下[3, 24]。另一方面，COPS 集群*之间的*复制是异步进行的，以确保客户端操作的低延迟和面对外部分区时的可用性。

系统组件。COPS 由两个主要软件组件组成：

- *键值存储。*COPS 的基本构件是一个标准键值存储库，可

对键进行线性化操作。COPS 以两种方式扩展了标准键值存储，COPS-GT 增加了第三种扩展。

1. 每个键值对都有相关的元数据。在 COPS 中，元数据是一个版本号。在 COPS-GT 中，它既是一个版本号，也是一个依赖关系（其他键及其各自的版本）列表。
2. 作为键值接口的一部分，键值存储还输出了三个额外的操作：按版本获取、后置和删除检查，下文将逐一介绍。这些操作支持 COPS 客户端库和异步复制流程，该流程支持因果关系+……。
   一致性和获取交易。
3. 对于 COPS-GT，系统会保留键值对的旧版本，而不仅仅是最新版本，以确保能够提供获取事务。第 4.3 节将进一步讨论保留旧版本的问题。

- *客户端库。*客户端库向应用程序输出两种主要操作：通过 `get`（在 COPS 中）或 `get trans`（在 COPS-GT 中）读取，以及通过 `put` 写入。[3]客户端库还通过客户端库应用程序接口（API）中的一个*文本*参数来保存客户端当前依赖关系的状态。

目标COPS 的设计旨在提供因果+一致性，其资源和性能开销与现有的最终一致性系统类似。因此，COPS 和 COPS-GT 必须

- *尽量减少一致性保护复制的开销。*COPS 必须确保以因果+一致的方式在群组间复制值。然而，一个简单的实现方法需要对一个值的所有依赖关系进行检查。我们提出了一种机制，利用因果依赖关系固有的图结构，只需要少量的此类检查。

[3] 本文对面向客户端的 API 调用（如获取）和数据存储 API 调用（如按版本获取）使用了不同的固定宽度字体。

- *(COPS-GT) 尽量减少空间需求。*COPS-GT 存储（可能）每个密钥的多个版本及其相关的依赖元数据。COPS-GT 使用积极的垃圾回收来清除旧状态（见第 5.1 节）。
- *(COPS-GT) 确保快速的获取传递操作。*COPS-GT 中的获取传递操作确保返回值的集合是因果+ 一致的（满足所有依赖关系）。天真的算法可能会阻塞和/或耗费无限多的获取回合来完成。这两种情况都与 ALPS 系统的可用性和低延迟目标不符。
  轮本地获取版本操作。

4.2 COPS 键值存储

与传统的（*key*，val）元组存储不同，COPS 必须跟踪写入值的版本，以及在以下情况下它们的依赖关系
的 COPS-GT。在 COPS 中，系统会存储每个密钥的最新版本号和值。在 COPS-GT 中，系统将每个密钥映射到一个版本条目列表，每个版本条目由（*版本*、
值，*deps*）。*deps* 字段是该版本的零个或多个
依赖关系；每个依赖关系都是一个（*鍵*、*版本*）对。
每个 COPS 集群都有自己的键值存储副本。
为了提高可扩展性，我们采用一致散列法[27]在集群节点间划分密钥空间，也可以采用其他技术（如基于目录的方法[6, 21]）。为了容错，每个密钥都会通过链式复制（chain replication）[5, 48, 51]复制到少量节点上。在集群中的节点之间，获取和投放都是可线性的。操作在*本地*集群中执行完毕后，会立即返回客户端库；集群间的操作是异步进行的。

COPS 中存储的每个密钥在每个群集中都有一个*主*节点。我们把一个密钥在所有群集中的主节点集合称为该密钥的*等效节点*。在实践中，COPS 的一致散列会让每个节点负责几个不同的密钥范围。在不同的数据中心，密钥范围可能有不同的大小和节点映射，但特定节点需要通信的等效节点总数与数据中心的数量成正比（也就是说，不同数据中心的节点之间的通信*并非*全对全）。

本地写入完成后，主节点会将其放入复制队列，然后异步发送到远程对等节点。这些节点则会等待，直到其本地集群满足了该值的依赖条件，才在本地提交

```
# Alice 的照片上传
ctx_id = createContext()    // Alice 登录
put(Photo, "Portuguese Coast", ctx_id)
put(Album, "add &Photo", ctx_id)
deleteContext(ctx_id)       // Alice 注销登录

# 鲍勃的照片视图
ctx_id = createContext() // 鲍勃登录
"&Photo" ← get(Album, ctx_id)
"葡萄牙海岸" ←get(Photo, ctx_id)
删除上下文（ctx_id）          // 鲍勃退出登录
```

图 5：第 3.2 节中使用 COPS 程序员界面上传照片的伪代码片段。使用 COPS-GT 时，每次获取都是对单个密钥的获取。

值。这种依赖性检查机制可确保写操作按照因果关系一致的顺序进行，读操作不会阻塞。

4.3 客户端库和界面

COPS 客户端库提供了一个简单直观的编程界面。图 5 演示了如何在照片上传场景中使用该界面。客户端 API 包括四种操作：

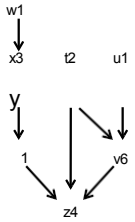
1. *ctx_id* ← `createContext`()
2. bool ← `deleteContext`(*ctx id*)
3. bool ← `put`（*key*、*value*、*ctx id*）
4. *值* ← 获取（*键*，*ctx id*）[在 COPS 中] 或 (*values*) ← `get trans` ((*keys*), *ctx id*) [COPS-GT 中] 4.

客户端 API 与传统的键值接口有两点不同。首先，COPS-GT 提供了 `get trans` 功能，一次调用即可返回多个键值对的一致视图。其次，所有函数都有一个上下文参数，库内部使用该参数来跟踪每个客户端操作的因果依赖关系[49]。上下文定义了 causal+ 的 "执行线程"。单个进程可能包含多个独立的执行线程（例如，并发服务于 1000 个独立连接的网络服务器）。通过分离不同的执行线程，COPS 可以避免因混合执行线程而产生的错误依赖关系。

接下来，我们将介绍 COPS-GT 中客户端库为确保获取事务的一致性而保存的状态。然后，我们将展示 COPS 如何大幅减少依赖状态的存储量。

COPS-GT 客户端库。COPS-GT 的客户端库将客户端的上下文存储在一个由（*key*、*version*、*deps*）条目组成的表中。客户端在应用程序接口中使用上下文 *ID*（*ctx id*）引用其*上下文*。[4]当

当客户端从数据存储中获取一个键时，库会将该键及其因果依赖关系添加到上下文中。当客户端放入一个值时，库会将



	最近数值	所有数值
t_{-2}		-
u_{-1}		-
$v\ t_6$	$_2, u\ t_1$	$_2, u_1$
w_{-1}		-
$x\ w\ w_3$	$_1$	$_1$
$y\ x\ x_1$	$_3$	$_3, w_1$
$z\ y_4$	$_1, v\ t_6$	$_2, u_1, v_6, w_1, x_3, y_1$

放入值的依赖关系设置为当前上下文中每个键的 最新版本。成功放入数据存储后，会返回分配给写入值的版本号 *v*。版本号然后，客户机库会将这个新条目（*key*, *v*, *D*）添加到上下文中。因此，上下文包括了之前读取或写入的所有值。

如图 6 所示，客户会话中的十个依赖关系，以及所有这些依赖关系的依赖关系。这就引起了对因果关系图潜在规模的两个担忧：(i) 存储这些依赖关系的状态要求，包括在客户端库和

图 6：客户因果关系示例图

上下文。箭头表示因果关系（例如，x_3 依赖于 w_1）。该表列出了每个值的所有依赖关系，以及用于最小化依赖关系检查的 "最近 "依赖关系。

(ii) 在集群间复制写入时必须进行的潜在检查次数，以确保因果一致性。为了减少跟踪依赖关系所需的客户端和数据存储状态，COPS-GT 提供了垃圾回收功能（详见第 5.1 节），一旦依赖关系被提交到所有 COPS 复制，就会被删除。

为减少复制过程中的依赖性检查次数，客户端库为服务器确定了几种可能的优化方法。考虑图 6 中的图。y_1 依赖于 x_3，并且根据传递性，依赖于 w_1。如果提交 y_1 的存储节点确定 x_3 已被提交，那么它就能推断出 w_1 也已被提交，因此无需明确检查。同样，虽然 z_4 直接依赖于 t_2 和 v_6，但提交节点只需检查 v_6，因为 v_6 本身依赖于 t。[2]

我们将*必须*检查的依赖关系称为*最近*依赖关系，列于图 6 的表格中。[5]为使服务器能够使用这些优化，客户端库首先会计算写入依赖列表中的最近依赖，并在发出写入时对其进行相应标记。

对于提供因果+ 一致性的键值存储来说，最近依赖关系就足够了；只有在 COPS-GT 中提供获取跨操作时，才需要完整的依赖关系列表。

COPS 客户端库。COPS 客户端库所需的状态和复杂度大大降低，因为它只需要学习最近的依赖关系，而不是所有的依赖关系。因此，它不会存储或检索它所获取的任何值的依赖关系：检索到的值比它的任何依赖关系都近，因此不需要依赖关系。

因此，COPS 客户端库只存储（*密钥*、*版本*）条目。对于获取操作，检索到的（*密钥*、*版本*）将被添加到上下文中。对于放入操作，库将当前上下文作为

最近的依赖关系，清除上下文，然后只用这个 put 重新填充上下文。这个 put 依赖于之前的所有密钥- 版本对，因此比它们更接近。

4.4 在 COPS 和 COPS-GT 中编写值

在介绍客户端库和键值存储的基础上，我们现在来了解一下向 COPS 写入值的步骤。COPS 中的所有写入都会首先进入客户端的本地集群，然后异步传播到远程集群。键值存储只需输出一个 API 调用，就能完成这两种操作：

(bool,*vers*) ' 放在 -(*key*、*val*、[*deps*]、*nearest*、vers=*θ*）之后

-

[4] 在库中保持状态并传递 ID 是一种设计。选择；也可以将整个上下文表编码为不透明 blob，并在客户端和程序库之间传递，这样程序库就无状态了。

[5] 用图论术语来说，一个值的最近依赖关系是因果关系图中长度为 1 的通向该值的*最长*路径。

写入本地集群。当客户端调用 put（*key,val,ctx id*）时，程序库会计算出完整的依赖关系图元集 *deps*，并将其中一些依赖关系图元识别为与值*最近的*图元。然后，程序库会在没有*版本*参数（即设置*版本=θ*）。在 COPS-GT 中，库在调用后的 put 中包含了 *deps*，因为依赖关系
必须与值一起存储；在 COPS 中，库只需 包含 *nearest*，而无需包含 *deps*。[6]本地集群中密钥的主存储节点会为密钥分配一个版本号，并将其返回给客户端库。我们限制每个客户端只能有一个未完成的输入；这是必要的，因为后面的输入必须知道前面输入的版本号，这样才能依赖它们。

put after 操作可确保只有*在*写完依赖列表中的所有条目*后，*才将 *val* 提交到每个群集。在客户端的本地集群中，这一属性自动成立，因为本地存储提供了线性化。(如果 *y* 依赖于 *x*，那么 put(*x*) 必须在 put(*y*) 发布之前提交）。但在远程集群中，情况并非如此，我们将在下文讨论。

主存储节点使用 Lamport 时间戳 [31] 为每次更新分配一个唯一的版本号。节点将版本号的高阶位设置为其 Lamport 时钟，低阶位设置为其唯一的节点标识符。通过 Lamport 时间戳，COPS 可以为每个密钥的所有写入推导出一个单一的全局顺序。这种顺序隐式地实现了 "最后写入者获胜 "的收敛冲突处理策略。COPS 还能显式地检测和解决冲突，我们将在第 5.3 节对此进行讨论。请注意，由于 Lamport 时间戳以尊重潜在因果关系的方式为所有分布式事件提供了部分排序，因此这种全局排序与 COPS 的因果一致性是兼容的。

群集间的写复制。在本地提交写操作后，主存储节点会使用 "put after "操作流将写操作异步复制到不同集群中的对等节点；不过，在这里，主节点会在 "put after "调用中包含密钥的版本号。与本地后置调用一样，COPS-GT 中包含 *deps* 参数，而 COPS 中不包含。这种方法扩展性很好，避免了对单个序列化点的需求，但要求接收更新的远程节点只有在更新的依赖节点提交到同一集群后才提交更新。

为确保这一特性，从另一个群集接收 "后置 "请求的节点必须确定该值的*最近*依赖关系是否已在本地得到满足。为此，它会向负责这些依赖关系的本地节点发出检查：

bool ' 深度检查（*密钥、版本*）

当节点收到 dep 检查时，它会检查其本地状态，以确定依赖关系值是否已被写入。如果是，它会立即响应操作。如果没有，它就会阻塞，直到所需的版本被写入。

如果对最近的依赖关系进行的所有 dep 检查操作都成功了，处理后置请求的节点就会提交写入的值，使其可用于本地集群中的其他读写操作。(如果任何 dep 检查操作超时，则处理后置请求的节点会重新发出该请求，如果出现故障，则有可能将该请求发送到新的节点。

4.5 在 COPS 中读取数值

与写入一样，读取也是在本地群集中完成的。客户端使用适当的上下文调用 `get` 库函数，而库则向本地集群中负责键的节点发出读取请求：

(值、版本、配置文件) ' 通过版本（*key*，version=LATEST）获取

这种读取可以请求密钥的最新版本，也可以请求特定的旧版本。请求最新版本等同于普通的单密钥获取；请求特定版本则是实现获取事务的必要条件。因此，COPS 中的按版本获取操作总是请求最新版本。收到响应后
客户端库将 (*key*,*version*[,*deps*]) 元组添加到客户端上下文，并将*值*返回给调用代码。*文件*只存储在
COPS-GT 中没有。

4.6 在 COPS-GT 中获取交易

COPS-GT 客户端库提供了一个获取跨接口，因为使用单键获取接口读取一组依赖键无法确保因果+一致性，即使数据存储本身是因果+一致的。我们通过扩展相册示例来演示这个问题，其中包括访问控制，爱丽丝首先将她的相册 ACL 更改为 " 仅限好友"，然后写了一段新的旅行描述，并向相册中添加了更多照片。读取爱丽丝相册的代码的自然实现（但不正确！）可能是：(1) 获取 ACL，(2) 检查权限，(3) 获取相册描述和照片。这种方法包含一个直接的 "从时间到检查到时间到使发生）。计算*最近*依存关系的方法使
值之前，确保*所有*依赖关系都已满足。

这反过来又确保了因果关系的一致性。

[6] 我们使用括号符号（[]）表示参数为可选参数；可选参数用于 COPS-GT，但不用于 COPS。

用 "的竞赛条件：当 Eve 访问相册时，她的 `get`(ACL) 可能会返回允许任何人（包括 Eve）读取的旧 ACL，但她的 `get`(album 内容）可能会返回 "仅限好友 "版本。

有一种 " 稻草人 " 式的解决方案是要求客户端按照因果关系的相反顺序执行单键 `get` 操作：如果客户端先执行 `get`(相册) ，再执行 `get`(ACL) ，就不会出现上述问题。然而，这种解决方案*也*是不正确的。试想一下，爱丽丝在更新相册后认为有些照片过于个人化，于是她(3) 删除了这些照片并重写了描述，然后(4) 再次打开 ACL 。这个 " 稻草人 " 有一个从检查时间到使用时间的不同错误，即 `get`(album) 获取的是私人相册，而随后的 `get`(ACL) 获取的是 " 公共 "ACL 。简而言之，ACL 和相册条目没有正确的规范排序。

相反，一个更好的编程接口可以让客户端获得多个密钥的因果+ 一致视图。实现这种保证的标准方法是在一个事务中读取和写入所有相关的密钥；然而，这需要为所有分组密钥设置一个序列化点，而 COPS 为提高可扩展性和简洁性避免了这一点。取而代之的是，COPS 允许独立写入密钥（元数据中有明确的依赖关系），并提供了一个获取跨操作来检索多个密钥的一致视图。

获取交易。要在一个因果关系+ 一致的条件下检索多个值，需要
客户机可以用所需的密钥集调用 `get trans` ，例如，`get trans`((ACL 相册))。根据
该获取事务可以返回不同的组合
的 ACL 和专辑，但从未（ACL $_{公}$, Albumpersonal）。

COPS 客户端库分两轮实现获取事务算法，如图 7 所示。在第一轮中，库向本地群集发出 *n_个*并发的按版本获取操作，客户端在获取事务中列出的每个密钥都有一个。因为

```
# @参数 keys        键值列表
# @param ctx_id context id
# @return values 值列表

function get_trans(keys, ctx_id):
  # 并行获取密钥（第一轮）
  键中的 k
    results[k] = get_by_version(k, LATEST)

  # 计算因果关系正确版本 (ccv)
  键中的 k
    ccv[k] = max(ccv[k], results[k].vers)
    for dep in results[k].deps
      if dep.key in keys
        ccv[dep.key] = max(ccv[dep.key], dep.vers)

  # 并行获取所需的 ccvs（第二轮）
  键中的 k
    if ccv[k] > results[k].vers
      results[k] = get_by_version(k, ccv[k])

  # 更新存储在上下文中的元数据
  update_context(results, ctx_id)

  # 只向客户端返回值
  return extract_values(results)
```

图 7：获取传输算法的伪代码。

COPS-GT 在本地提交写入，本地数据存储保证这些明确列出的每个键的依赖关系都已满足，也就是说，它们已在本地写入，对它们的读取会立即返回。不过，这些明确列出的、独立再提取的值之间可能并不一致、
如上所示。每次按版本获取操作都会返回一个（*value*、*version*、*deps*）元组，其中 *deps* 是键和版本的列表。然后，客户端库会检查每个依赖项（*键*、*版本*）。如果该结果的
客户端没有请求依赖密钥，或者如果请求了，它检索到的版本≥ 依赖关系列表中的版本。

对于所有未满足要求的密钥，图书馆会发出第二轮
的并发版本获取操作。请求的版本将是在任何依赖关系列表中看到的最新版本。
第一轮。这些版本满足第一轮的所有因果依赖关系，因为它们≥ 所需的版本。此外，由于依赖关系是传递性的，而且这些第二轮版本
都依赖于第一轮检索的版本，因此它们不会引入任何需要满足的新依赖关系。这种算法允许 `get trans` 返回数据存储在第一轮检索的最新时间戳时的一致视图。

只有当客户端必须读取比第一轮获取的版本更新的版本时，才会进行第二轮读取。只有在获取事务中涉及的键在第一轮中更新时，才会出现这种情况。因此，我们认为第二轮很少发生。在我们的例子中，如果夏娃与爱丽丝的写入同时发出获取事务，那么算法的第一轮获取可能会检索到公共 ACL 和私人相册。然而，私人相册取决于 "仅限好友 "的 ACL，因此第二轮将获取 ACL 的最新版本，从而让 `get trans` 向客户端返回一组因果+一致的值。

数据存储的因果+一致性为获取事务算法的第二轮提供了两个重要特性。首先，按版本获取请求会立即成功，因为重新请求的版本必须已经存在于本地集群中。其次，新的按版本获取请求不会引入任何新的挂起点，因为这些挂起点在第一轮请求中就已经知道了。

由于反向性，在第二轮交易中，交易将被指定为最新版本。第二个特性说明了为什么获取事务算法在第二轮中指定了一个明确的版本，而不仅仅是获取最新版本：否则，在并发写入的情况下，一个更新的版本可能会引入更多新的依赖关系，这可能会无限期地持续下去。

5. 垃圾、故障和冲突

本节将介绍 COPS 和 COPS- GT 的三个重要方面：它们的垃圾收集子系统（可减少系统中的额外状态）；它们针对客户端、节点和数据中心故障的容错设计；以及它们的冲突检测机制。

5.1 垃圾回收子系统

COPS 和 COPS-GT 客户端会存储元数据；COPS-GT 服务器会额外保存多个版本的密钥和依赖关系。如果不进行干预，随着密钥的更新和插入，系统的空间占用会无限制地增长。COPS 垃圾回收子系统会删除不需要的状态，使系统总大小保持在可控范围内。第 6 节评估了维护和传输这些额外元数据的开销。

版本垃圾回收。(仅限 COPS-GT）。
*存储的内容*COPS-GT 可存储每个密钥的多个版本，以满足客户端按版本获取密钥的请求。
为什么可以清理： `get trans` 算法限制了完成一个 get 事务所需的版本数量。该算法的第二轮仅对晚于第一轮返回的版本发出按版本获取的请求。为了能及时进行垃圾回收，COPS-GT 通过一个可配置的参数--*trans time*（在我们的实现中设置为 5 秒）来限制 `get trans` 的总运行时间。(如果超时，客户端库将重新启动 `get trans` 调用，并使用较新版本的密钥满足事务；我们预计只有在集群中多个节点崩溃时才会发生这种情况）。
*何时可以清理：在*写入密钥的新版本后，COPS-GT 只需在*跨时间*内保留旧版本，再加上一小段时间的时钟偏移。过了这段时间，就不会再有版本获取调用请求旧版本，垃圾回收器就可以将其删除。
空间开销：空间开销受*跨时间*内可创建的旧版本数量限制。这个数字由节点所能承受的最大写入吞吐量决定。我们的实施方案支持每个节点每秒 105MB 的写入流量，因此需要（可能）额外 525MB 的缓冲空间来保存旧版本。这种开销是按机器计算的，不会随着集群规模或数据中心数量的增加而增加。

依赖关系垃圾回收。(仅限 COPS-GT）。
*存储的内容*存储依赖关系是为了让获取事务获得数据存储的一致视图。
*为什么可以清理：*一旦不再需要与旧依赖关系相关的版本来保证获取事务操作的正确性，COPS-GT 就可以对这些依赖关系进行垃圾回收。

为了说明获取事务操作何时不再需要去挂起，请考虑依赖于 x_2 和 y_2 的值 z_2。x、y 和 z 的获取事务要求，如果返回了 z_2，则 $x_{\geq 2}$
和 $y_{\geq 2}$ 也必须返回。因果一致性确保
x_2 和 y_2 写在 z_2 之前。因果+一致性
渐进属性确保一旦写入 x_2 和 y_2，就会通过获取

操作。因此，一旦 z_2 被写入所有数据中心，且*跨时间*已过，任何返回 z_2 的 get 事务都将返回 $x_{\geq 2}$ 和 $y_{\geq 2}$ ，*这样* z_2 的依赖关系就可以被垃圾回收。

*何时可以清理：*在数值被清除后*的*几秒后

在*所有*数据中心都已提交的情况下，COPS-GT 可以清理值的依赖关系。(COPS 和 COPS-GT 都可以进一步设置值的*永不依赖*标志，这将在下文讨论。为了清除依赖关系，每个远程数据中心都会在写入已提交且超时时间已过时通知发起数据中心。一旦所有数据中心都确认，发起数据中心就会清除自己的依赖关系，并通知其他数据中心也这样做。为了最大限度地减少用于清理依赖关系的带宽，只有当某个密钥的该版本是*跨时间*秒后的最新版本时，副本才会通知始发数据中心；如果不是，则无需收集依赖关系，因为整个版本都会被收集。[7]

*空间开销：*在正常运行情况下，依赖项会在*传输时间*加上往返时间后被垃圾回收。依赖关系只对最新版本的密钥进行收集，旧版本的密钥已经如上所述进行了垃圾收集。

在分区期间，无法收集对最新版本密钥的依赖性。这是 COPS-GT 的一个限制，尽管我们认为长时间的分区很少见。为了说明为什么这种让步对于获得事务的正确性是必要的，请考虑依赖于值 a_2 的值 b_2 ：如果过早地记录了 b_2 对 $_2$ 的依赖关系，那么后面的值 c_2 在因果关系上依赖于 b_2 ，因此也依赖于 a_2 ，它可以在不明确依赖于 a_2 的情况下被写入。那么，如果 a_2 、b_2 和 c_2 都在短时间内被复制到数据中心，那么第一轮获取事务就可以获得 a_1 、a 的早期版本和 c_2 ，然后将这两个值返回给客户端，这正是因为它不知道 c_2 依赖于较新的 a 。[2]

客户端元数据垃圾回收。(COPS + COPS-GT）*存储的内容：*COPS 客户端库使用与所有操作一起传递的 *ctx id* 跟踪客户端会话（单线程执行）期间的所有操作。与上面讨论的存在于键值存储中的依赖关系信息不同，这里讨论的依赖关系是客户端元数据的一部分，并存储在客户端库中。在这两个系统中，自上次 "放入 "后的每次 "获取 "都会增加一个最近的依赖关系。此外，在 COPS-GT 中，所有在获取操作中返回的新值及其依赖关系，以及所有放入操作都会添加正常依赖关系。如果客户端会话持续很长时间，更新所附依赖关系的数量就会增加，从而增大 COPS 需要存储的依赖关系元数据的大小。

*为什么可以清理：*与上面的依赖关系跟踪一样，客户只需跟踪依赖关系，直到保证它们在任何地方都能得到满足。

*当可以清理时：*COPS 通过两种方式减少客户端状态（上下文）的大小。首先，如上文所述，一旦一个 put 版本成功提交到所有数据中心，COPS 就会将该关键版本标记为 "*从不依赖*"（never-*depend*），以表明客户端无需对其表示依赖。此外，这一过程是传递性的：*从不依赖项所依赖的*任何内容都必须被标记为*从不依赖，*因此它也可以从上下文中被垃圾回收。

其次，COPS 存储节点会从put after 操作中删除不必要的依赖项。当节点接收到放盘后操作时，它会检查依赖关系列表中的每个项目，并删除版本号早于*全局检查点时间*的项目。该检查点时间是整个系统中*所有*节点都满足的最新 Lamport 时间戳。COPS 键值存储库会将此检查点时间返回给客户端库（例如，在响应 put after 时），以便客户端库从上下文中清除这些依赖关系。[8]

要计算全局检查点时间，每个存储节点首先要确定它作为主节点的密钥范围内任何待处理密钥的最旧 Lamport 时间戳。(换句话说，它要确定所有副本中不能保证满足的最旧密钥的时间戳）。然后，它联系其他数据中心的对等节点（处理相同密钥范围的节点）。这些节点成对地交换它们的最短 Lamport 时间，同时记住任何副本中观测到的最旧 Lamport 时钟。完成这一步骤后，所有数据中心都拥有相同的信息：每个节点都知道其密钥范围内全球最旧的 Lamport 时间戳。然后，数据中心内的节点围绕每个范围内的最小时间戳进行闲聊，找出其中任何一个节点观察到的最小 Lamport 时间戳。在我们的实施过程中，这个周期性过程每秒进行 10 次，对性能没有明显影响。

5.2 容错

COPS 可抵御客户端、节点和数据中心故障。在下面的讨论中，我们假定故障是 "故障停止 "的：组件停止响应故障，而不是错误或恶意运行，并且故障是可检测的。

客户端故障。COPS 的键值接口意味着每个客户端请求（通过库）都由数据存储系统独立原子处理。从存储系统的角度看，如果客户端出现故障，它只需停止发出新请求，无需恢复。从客户端的角度来看，COPS 的依赖性跟踪通过确保参照完整性等属性，可以更轻松地处理其他客户端的故障。请看照片和相册的例子：如果客户端在写入照片后但在写入照片引用前发生故障，数据存储仍将处于一致状态。如果照片本身没有被写入，就永远不会有照片引用的实例。

键值节点故障。COPS 可以使用任何底层容错线性化键值存储。我们的系统建立在 FAWN-KV [5] 节点的独立集群之上，集群*内*使用链式复制 [51] 来掩盖节点故障。因此，我们将介绍 COPS 如何使用链式复制来提供节点故障容错。

与 FAWN-KV 的设计类似，每个数据项都存储在由 R 个连续节点组成的一致散列环链中。

服务器发布的操作涉及的内容更多一些，因为它们是由不同的节点链发布和处理的。本地集群中的尾部会向每个远程数据中心的头部复制 "put after " 操作。然后，远程数据中心的数据头会将检查 的操作（本质上是读操作）发送到相应的服务器。

[7] 我们目前正在研究是否要以这种方式收集依赖关系。与在全局检查点时间后收集这些信息相比（下文将讨论），"全局检查点时间 "提供了足够大的好处，足以证明其信息传递成本是合理的。

[8] 由于未完成读取，客户端和服务器还必须等待在使用新的全局检查点时间之前，它们需要*跨时间*秒。

尾部。一旦这些操作返回（如果操作未返回，则会超时，并重新进行 dep 检查），远程头部就会将值沿着（远程）链向下传播到远程尾部，而远程尾部则会提交该值，并向发起数据中心确认该操作。

依赖关系垃圾收集遵循类似的互锁链模式，但为简洁起见，我们省略了细节。版本垃圾收集在每个节点上本地进行，与单节点情况下的操作一样。对于客户端元数据垃圾收集，全局检查点时间的计算正常进行，每个尾部都会更新其相应的键范围最小值。

数据中心故障。COPS 的分区容错设计还提供了对整个数据中心故障（或分区）的恢复能力。面对此类故障，COPS 仍可正常运行，但有一些关键区别。

首先，源于故障数据中心但尚未复制出去的任何后置操作都将丢失。这是允许低延迟本地写入的必然代价，因为本地写入的返回速度要快于数据中心之间的传播延迟。如果数据中心仅被分区且未发生故障，则不会丢失写入数据。相反，它们只会延迟到分区恢复。[9]

其次，运行中数据中心的复制队列所需的存储空间会增加。它们将无法向故障数据中心发送 "put after "操作，因此 COPS 将无法对这些依赖关系进行垃圾回收。系统管理员有两个选择：如果分区可能很快愈合，则允许队列增长；或者重新配置 COPS，使其不再使用故障数据中心。

第三，在 COPS-GT 中，面对数据中心故障，依赖性垃圾收集无法继续进行，直到分区愈合或系统重新配置以排除故障数据中心。

5.3 冲突探测

当有两个 "同时"（即不在同一上下文/执行线程中）写入给定密钥时，就会发生冲突。COPS 系统默认采用 "最后写入-获胜 "策略来避免冲突检测。最后 "写入是通过比较版本号来确定的，这样可以避免冲突检测，提高简洁性和效率。我们相信这种行为对许多应用都很有用。不过，在其他一些应用中，采用更明确的冲突检测方案会使推理和编程变得更简单。对于这些应用，可以将 COPS 配置为检测冲突操作，然后调用一些特定于应用的收敛冲突处理程序。

带有冲突检测功能的 COPS（我们称之为 COPS-CD）为系统增加了三个新组件。首先，所有写入操作都会携带*先前版本*元数据，该元数据会显示写入时本地集群可见的密钥的最新先前版本（该先前版本可能为空）。其次，现在所有写入操作都隐式依赖于之前的版本，这确保了新版本只能在其之前的版本之后写入。这种隐式依赖关系需要额外的 dep 检查操作，不过这种操作的开销很小，而且总是在本地机器上执行。第三，COPS-CD 有一个由应用程序指定的聚合冲突处理程序，在检测到冲突时会调用它。

[9] 在因果+模型中支持提交所需的数据中心数量的灵活性仍是未来工作的一个有趣方面。

系统	因果+	可	扩展Get Trans
日志	是	没有	没有
COPS	是	是	没有
COPS-GT	是	是	是

表 1：三个比较系统的摘要。

COPS-CD 遵循一个简单的程序来确定一个密钥的*新*操作（以前的版本 *prev*）是否与该密钥当前的可见版本 *curr* 相冲突：

prev ≠ *curr*，当且仅当 *new* 和 *curr* 冲突时。

我们省略了完整的证明，但在此介绍一下直观感受。在前进方向上，我们知道 *prev* 必须写在 *new* 之前，*prev* ≠ *curr*，而且要使 *curr* 代替 *prev* 可见，我们必须根据因果+ 的进展属性使 *curr* > *prev*。但是，由于 *prev* 是 *new* 在因果关系上的最新版本，我们可以
得出 *curr* ∕∼ *new* 的结论。此外，由于 *curr* 写于 *new* 之前，因此它不可能在 *new* 之后，所以 *new* ∕∼ *curr* 两者冲突。反过来，如果 *new* 和 *curr* 冲突，那么 *curr* ∕∼ *new*。根据定义，*prev* ∼ *new*，因此 *curr* ≠ *prev*。

6. 评估

本节将介绍 COPS 和 COPS-GT 的评估情况，包括使用微基准建立基准系统延迟和吞吐量，进行敏感性分析以探索不同参数对动态工作负载的影响，以及进行更大规模的端到端实验以展示可扩展的因果+一致性。

6.1 实施与实验设置

COPS 约有 13000 行 C++ 代码。它建立在 FAWN-KV [5, 18] (~8500 LOC) 的基础上，后者在本地集群内提供线性化键值存储。COPS 使用 Apache
Thrift [7] 用于所有系统组件之间的通信，谷歌的 Snappy [45] 用于压缩依赖列表。我们目前的原型实现了论文中描述的所有功能，但不包括用于本地容错和冲突检测的链式复制。[10] 和冲突检测。

我们对三个系统进行了比较：LOG、COPS 和 COPS-GT。LOG 使用 COPS 代码库，但排除了所有依赖关系跟踪，从而模拟了之前使用日志交换建立因果一致性的工作（如 Bayou [41] 和 PRACTI [10]）。表 1 总结了这三个系统。

每个实验都在 VICCI 试验平台 [52] 的一个集群上运行。集群的 70 台服务器为用户提供了一个隔离的 Linux VServer。每台服务器都有 2x6 核 Intel Xeon X5650 CPU、48GB 内存、3x1TB 硬盘和 2x1GigE 网络端口。

在每次实验中，我们都会根据需要将集群划分为多个逻辑 "数据中心"。我们保留了不同数据中心节点之间的全部带宽，以反映它们之间通常存在的高带宽骨干网。在 FAWN-KV 中，所有读写操作都会进入磁盘，但我们实验中的大多数操作都会进入内核缓冲缓存。

所展示的结果来自 60 秒的试验。每次试验的前 15 秒和后 15 秒的数据被省略，以避免出现实验假象，同时也让垃圾收集和复制机制得以加速。我们每次试验运行 15 次，并报告中位数；最小和最大结果几乎总是在 6% 以内。

[10] 在我们的实施所基于的 FAWN-KV 版本中，链式复制功能并不完善。

系统	运行	延迟（毫秒） 50%	99%	99.9%	吞吐量 (
节俭	ping	0.26	K6β̃s/s)	12.25	60
	COPSget 按版本	0.37	3.08	11.29	52
	COPS-GTget 按版本 52		0.38	3.14	9.52
	COPSput 后 (1)	0.57	6.91	11.37	30
	COPS-GTput 后(1)	0.91	5.37	7.37	24
	COPS-GTput 之后 (130) 20		1.03	7.45	11.54

表 2：不同的数据传输延迟（以毫秒为单位）和吞吐量（以 Kops/秒为单位）。

put after(*x*) 包含 *x* 依赖关系的元数据。

我们将吞吐量差异较大的少数试验归因于 VICCI 平台的共享性。

6.2 微基准测试

我们首先在一个简单的环境中评估 COPS 和 COPS-GT 的性能特征：两个数据中心，每个数据中心一台服务器，一台同地客户端机器。客户机向其本地服务器发送put 和 get 请求，试图使系统达到饱和。请求分布在 2^{18} 个密钥上，有 1B 个值--我们使用 1B 值是为了与后面的实验保持一致，在后面的实验中，小值是 COPS 的最坏情况（见图 11(c)）。表 2 显示了延迟和吞吐量的中位数、99% 和 99.9%。

COPS 在设计上决定以宽松的方式处理客户端操作，从而降低了所有操作的延迟。COPS 和 COPS-GT 中按版本获取操作的延迟与使用 Thrift 的端到端 RPC ping 相似。放操作的延迟稍高，因为它们的计算成本更高；它们需要更新元数据和写入值。不过，"put after "操作的延迟中位数约为 1 毫秒，即使那些依赖关系多达 130 个的操作也是如此。

系统吞吐量也遵循类似的模式。按版本获取操作实现了高吞吐量，与 Thrift ping 操作类似。erations（52 对 60 Kops/s）。COPS 服务器处理put after 操作的速度为 30 Kops/s（此类操作的计算成本高于 get 操作），而 COPS-GT 在put after 操作有 1 个依赖项时的吞吐量要低 20%（这是由于垃圾收集旧版本的成本所致）。随着 COP-GT 后放操作的依赖项数量增加，吞吐量会下降

由于每个操作中的元数据较大（每个依赖项约为 12B），因此略有增加。

6.3 动态工作量

我们对访问 COPS 系统的交互式客户机的动态工作负载建模如下。我们建立了两个数据中心，每个数据中心有 *S 台*服务器

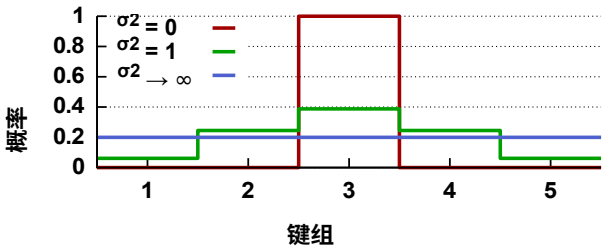


图 8：在我们的实验中，客户端在选择访问密钥时，首先根据某种正态分布选择一个密钥组，然后根据均匀分布在该组中随机选择一个密钥。图中显示了 3 号客户端（共 5 个）在不同方差下的阶跃正态分布。

选择每个操作的密钥是为了控制操作之间的依赖程度（即从完全隔离到完全混合）。具体来说，给定 *N 个*客户端，整个密钥空间由 *N 个*密钥组（R_1 ... ...R_N，每个客户一个。每个密钥组包含 *K 个密钥*，这些密钥是随机分布的（也就是说，它们并不都在同一个服务器上）。客户端在进行操作时，按以下步骤选择密钥。首先，它们从定义在 *N 个密钥*组上的正态分布（每个密钥组的宽度为 1）中采样，选出一个密钥组。然后，它们在该密钥组内均匀随机地选择一个密钥。结果是密钥的分布，同一密钥组内的密钥可能性相等，不同密钥组的可能性可能不同。

图 8 举例说明了 3 号客户机（共有 5 个客户机）在方差为 0、1 和极限接近 ∞ 时的密钥组分布情况。当方差为 0 时，客户端将限制访问
自己的 "密钥组"，从不与其他客户交互。相比之下
当方差 → ∞ 时，客户端的访问在所有密钥上均匀随机分布，从而导致操作后投放之间的相互依赖性最大。

除非另有说明，动态工作量实验的参数如下：

参数	默认值	参数	默认值
数据中心	2	投放：获取比例1 或 1:4	:1
服务器/数据中心	4		
客户端/服务器	1024	差异	1
键/键组	512	数值大小	1B

，在其中一个数据中心放置 *S 台*客户机。客户机访问本地数据中心的存储服务器，本地数据中心会将操作后的数据传输到远程数据中心。我们在实验中报告的是*可持续*吞吐量，即两个数据中心都能处理的最大吞吐量。在大多数情况下，本地数据中心的 COPS 会占用 CPU，而远程数据中心的 COPS-GT 也会占用 CPU。

为了更好地减轻系统压力，更准确地描述实际运行情况，每台客户机都模拟了多个逻辑 COPS 客户机。

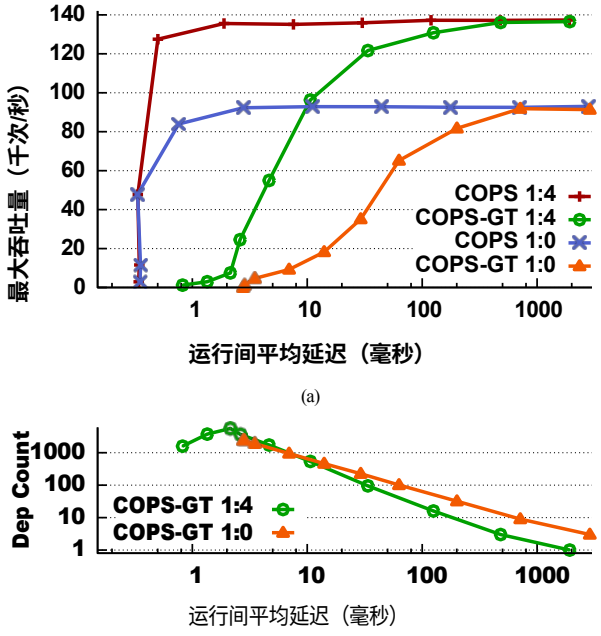
客户端每次执行操作时，都会随机执行一个 put 或获取操作。特定实验中的所有操作都使用固定大小的值。

由于这些变量所有可能组合的状态空间为
因此，下面的实验将逐个探究各项参数。

每服务器客户端我们首先描述了系统吞吐量与客户端操作之间延迟增加的函数关系（针对两种不同的put:get比率）。[11]图 9(a) 显示，当操作间延迟较低时，COPS 明显优于 COPS-GT。相反，当操作间延迟接近几百毫秒时，COPS 和 COPS-GT 的最大吞吐量趋于一致。图 9(b) 有助于解释这种行为：随着操作间延迟的增加，由于垃圾收集的持续进行，每个操作的依赖性数量会*减少*。

[11] 在这些实验中，我们没有直接控制运行间延迟。相反，我们将每台客户机上运行的逻辑客户机数量从 1 个增加到 2 个[18]；在我们的测试框架中，客户机的线程池大小固定，因此每个逻辑客户机的调度频率较低。由于每个客户机都是先请求一次，然后才退出，因此平均延迟会更高。

操作间延迟（简单计算为$\frac{\text{吞吐量}}{\text{客户/服务器数量}}$）。我们的默认设置为 1024 放取比为 1:0，1:0 的 COPS 为 11ms，1:4 的 COPS-GT 为 11ms，1:4 的 COPS 为 8ms。



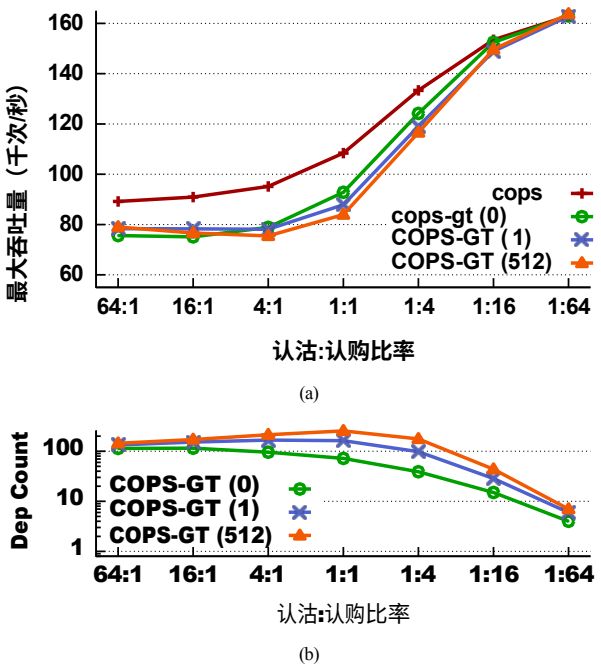


图 9：COPS 和 COPS-GT 的最大吞吐量和由此产生的平均等待时间大小（同一逻辑客户端的连续操作之间的输入间隔为给定值）。图例给出了投入与获取的比例（即 1:0 或 1:4）。

要理解这种关系，请看下面的示例。如果全局检查点时间比当前时间晚 6 秒，而逻辑客户机正在执行 100 次/秒的输入（全输入工作负载）、
每项投入将有 100 - 6 = 600 个依赖关系。图 9(b) 说明了这种关系。虽然 COPS 只存储单个最近的
依赖关系（未显示），COPS-GT 必须跟踪所有未被垃圾回收的依赖关系。这些额外的依赖关系解释了 COPS-GT 的性能：当输入间时间较小时，每个值都需要传播大量的依赖关系，因此每次操作的成本都较高。

全局检查点时间通常比当前时间滞后约 6 秒，因为它包含了*跨*时间延迟（每 Sec-
5.1 节）以及在本地数据中心周围闲聊检查点所需的时间（节点每 100 毫秒闲聊一次）。回想一下，为了确保当前正在执行的获取*传输*操作能够完成，需要有一个商定的传输时间延迟，而存储节点则使用闲谈来确定最老的未提交操作（因此也是可以进行垃圾回收的依赖关系的最新时间戳）。值得注意的是，数据中心*之间的*往返时间延迟只是滞后的一小部分，因此性能不会受到 RTT 的显著影响（例如，70 毫秒的广域 RTT 大约是全球检查点时间 6 秒滞后的 1%）。

投入：获取比例。接下来，我们评估了在不同的 Put:get 比例和密钥访问分布情况下的系统性能。图 10(a) 显示了 COPS 和 COPS-GT 在 64:1 到 1:64 的放取比和三种不同分布差异下的吞吐量。我们发现，读取较多的工作负载（put:get 比率<1）的吞吐量会增加，而 COPS-GT 在读取最多的工作负载上与 COPS 相比更具竞争力。虽然 COPS 在不同差异下的性能相同，但 COPS-GT 的吞吐量*会*受到差异的影响。我们通过分析 COPS 和 COPS-GT 之间的关系来解释这两种行为。

图 10：COPS 和 COPS-GT 在给定的 put:get 比例下的最大吞吐量和由此产生的平均等待时间。图例给出了方差（即 0、1 或 512）。

图 10(b)）；依赖性越少，需要处理的元数据就越少，吞吐量也就越高。

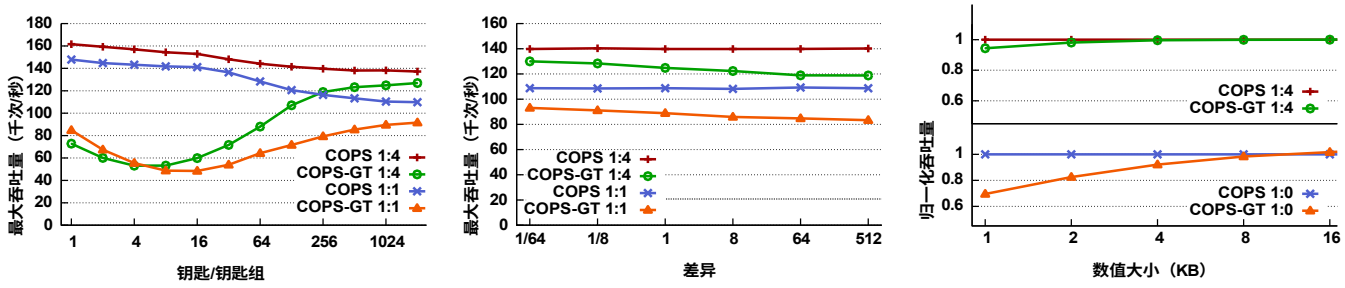
当不同的客户端访问相同的密钥时（方差 > 0），我们在图 10(b) 中观察到两个不同的阶段。首先，当put:get 的比例从 64:1 下降到 1:1，依赖关系的数量就会*增加*。之所以会出现这种增长，是因为每次获取操作都会增加客户端通过获取最近被其他客户端放入的值而继承新依赖关系的可能性。例如，如果客户机$_1$ 投放了一个依赖关系为 d 的值 v_1 ，而客户机$_2$ 读取了该值，那么客户机$_2$ 未来的投放将同时依赖于 v_1 和 v 。

d.其次，当 Put:get 的比例从 1:1 降到 1:64 时，依赖关系的数量就会*减少，*原因有两个：(i) 每个客户端执行的 Put 操作次数减少，因此每个值依赖于该客户端之前写入的值的次数减少；(ii) 由于 Put 操作次数减少，更多的键值比全局检查点时间更早，因此获取这些键值不会带来额外的依赖关系。

当客户端访问独立的密钥（方差 = 0）时，依赖关系的数量会随着输入：输出的比例严格递减。这一结果是意料之中的，因为每个客户端只访问自己键组中的值，而这些值是它之前写入并已经依赖的。因此，任何 get 都不会导致客户端继承新的依赖关系。

COPS 的平均依赖关系数（未显示）总是很低，从 1 个到 4 个不等，因为 COPS 只需要跟踪最近的（而不是所有的）依赖关系。

每个密钥组的密钥。图 11(a) 显示了密钥组大小对 COPS 和 COPS-GT 吞吐量的影响。回顾一下，客户端将其请求均匀分布到所选密钥组中的密钥上。COPS-GT 的行为有细微差别；我们通过考虑获取操作继承新依赖关系的可能性来解释其不同的吞吐量，这反过来又会降低吞吐量。在默认方差为 1 和密钥数/密钥组较少的情况下，多数



(a) (b) (c)

图 11：不同密钥/密钥组、差异和值大小下的最大系统吞吐量（使用 1:4、1:1 或 1:0 的 put:get 比率）。

客户端只能访问少量的键。一旦检索到一个值并继承了它的依赖关系，随后对同一值的获取不会导致客户端继承任何新的依赖关系。不过，随着键/键组的数量开始增加，客户端重复获取相同值的可能性就会降低，并开始继承额外的依赖关系。不过，随着键/键组数量的不断增加，垃圾回收也开始产生影响：越来越少的获取会检索到最近由其他客户端写入的值（例如，在全局检查点时间之后），因此越来越少的获取会返回新的依赖关系。COPS-GT 的性能呈现弓形，很可能就是这两种截然不同的影响造成的。

差异。图 11(b) 显示了差异对系统性能的影响。如前所述，COPS 的吞吐量不受不同差异的影响：COPS 中的获取操作从不继承额外的依赖关系，因为根据定义，返回值总是 "较近 " 的。然而，随着差异的增加，COPS-GT 继承依赖关系的几率也会增加，从而导致吞吐量下降。

数值大小。最后，图 11(c) 显示了数值大小对系统性能的影响。在本实验中，我们将系统的最大吞吐量与 COPS 的最大吞吐量进行了归一化处理（COPS 正好为 1.0 的数据线仅用于比较）。随着数值大小的增加，COPS-GT 的相对吞吐量接近 COPS 的吞吐量。

我们认为这有两个原因。首先，与处理实际值相比，处理依赖项（大小固定）的相对成本会降低。其次，随着每次操作处理时间的增加，操作间延迟也会相应增加，这反过来又会导致依赖关系的减少。

6.4 可扩展性

为了评估 COPS 和 COPS-GT 的可扩展性，我们将它们与 LOG 进行了比较。LOG 模拟基于日志序列化和交换的系统，它只能提供单节点复制的因果+一致性。我们的 LOG 实现使用 COPS 代码，但不包括依赖性跟踪。

图 12 显示了 COPS 和 COPS-GT（在 1、2、4、8 或 16 台服务器/数据中心上运行）与 LOG（在 1 台服务器/数据中心上运行）的吞吐量归一化。除非另有说明，所有实验均使用第 6.3 节中给出的默认设置，包括 1:1 的 put:get 比例。在所有实验中，在单台服务器/数据中心上运行的 COPS 的性能几乎与 LOG 相同。(毕竟，与 LOG 相比，COPS 只需要跟踪少量的数据。远程数据中心中的任何检查操作都可以作为本地函数调用来执行）。

更重要的是，我们发现 COPS 和 COPS-GT 在所有情况下都能很好地扩展：当我们将每个数据中心的服务器数量增加一倍时，吞吐量几乎翻了一番。

在使用所有默认设置的实验中，COPS 和 COPS-GT 的吞吐量虽然只有 COPS 的三分之二，但相对于它们自身而言，COPS 和 COPS-GT 的扩展性都很好。这些结果表明，选择默认参数是为了给系统提供一个非理想的工作负载。不过，在一些不同的条件下--事实上，在互联网服务中更为常见的工作负载下，COPS 和 COPS-GT 的性能几乎完全相同。

例如，当操作间延迟较高时，COPS-GT 的相对吞吐量接近 COPS 的吞吐量（通过每台服务器托管 32K 客户端实现，而不是默认的 1024 客户端；见脚注 11）。同样，在控制所有其他参数的情况下，读取量更大的工作负载（put:get 比为 1:16 对 1:1）、客户端访问分布差异更小（1/128 对 1）或更大的值（16KB 对 1B）都会产生类似的效果：COPS-GT 的吞吐量变得与 COPS 相当。

最后，在 "预期工作负载 "实验中，我们设置了更接近社交网络等互联网服务的参数。与默认值相比，该工作负载具有更高的操作间延迟（32K 客户端/服务器）、更大的值（1KB）和重读分布（1:16 比例）。在这些条件下，COPS 和 COPS-GT 的吞吐量非常接近，并且都能随着服务器数量的增加而扩展。

7. 相关工作

我们将相关工作分为四类：ALPS 系统、因果一致系统、可线性化系统和事务系统。

ALPS 系统。越来越多的 ALPS 系统包括最终一致性键值存储，如 Ama- zon 的 Dynamo [16]、LinkedIn 的 Project Voldemort [43]，以及流行的 memcached [19]。Facebook 的 Cassandra [30] 可以配置为使用最终一致性来实现 ALPS 特性，也可以牺牲 ALPS 特性来提供线性化。雅虎的 PNUTS [15] 对我们的工作产生了重要影响，它提供了按键顺序一致性（尽管他们将其命名为按记录时间轴一致性）。不过，PNUTS 并不提供任何键之间的一致性；实现这种一致性会带来 COPS 所要解决的可扩展性挑战。

因果一致性系统。以前的许多系统设计者都认识到因果一致性的实用性。Bayou [41] 为单机复制提供了一个类似 SQL 的接口，可实现因果+ 一致性。Bayou 可在本地处理所有读写操作；但它并不能实现我们所考虑的可扩展性目标。

TACT[53]是一个因果+一致系统，它使用顺序和数字约束来限制系统中复制的分歧。ISIS [12] 系统利用虚拟同步概念 [11]，为应用程序提供因果广播原语（CBcast）。

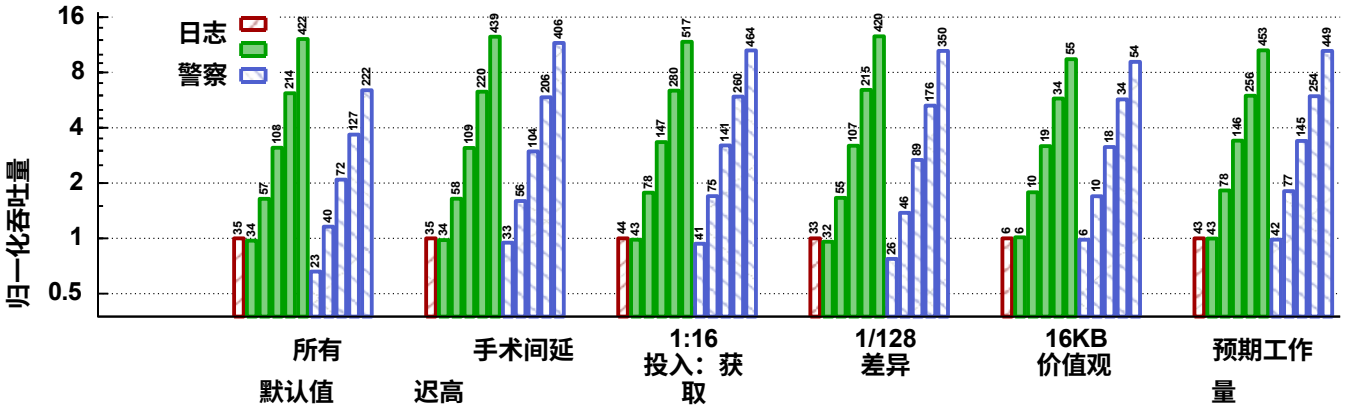


图 12：在各种情况下，1 台服务器/数据中心的 LOG 以及 1、2、4、8 和 16 台服务器/数据中心的 COPS 和 COPS-GT 的吞吐量。每种情况下的吞吐量与 LOG 的吞吐量进行了归一化处理；每个条形图上方给出了原始吞吐量（Kops/s）。

CBcast 可以直接用于提供因果一致的键值存储。通过因果记忆[2]共享信息的副本也能提供因果一致的 ALP 键值存储。不过，这些系统都需要单机复制，因此不具备可扩展性。

PRACTI [10] 是一个支持部分复制的因果+一致性 ALP 系统，它允许副本只存储密钥的子集，从而提供了一定的可扩展性。不过，每个副本--也就是提供因果+一致性的密钥集--仍然受限于单台机器所能处理的范围。

懒惰复制[29]最接近 COPS 的方法。懒复制明确标记了更新的因果依赖关系，并等待这些依赖关系得到满足后再在副本中应用。这些依赖关系通过前端进行维护并附加到更新上，前端类似于我们的客户端库。不过，懒复制的设计假定副本仅限于一台机器：每个副本都需要一个单点，该单点可以：(i) 创建所有副本操作的顺序日志；(ii) 将该日志传给其他副本；(iii) 将其操作日志与其他副本的日志合并；最后 (iv) 按因果顺序应用这些操作。最后，在并行理论工作中，Mahajan 等人 [35] 定义了实时因果（RTC）一致性，并证明它是始终可用系统中可实现的最强一致性。RTC 比 causal+ 更强，因为它强制要求实时性：如果因果并发写入不实时重叠，则较早的写入不得在较晚的写入之后排序。这种实时性要求有助于捕捉隐藏在系统中的潜在因果关系（如带外消息传递 [14]）。相比之下，causal+ 没有实时性要求，因此可以更高效地实现。值得注意的是，COPS 高效的 "最后写入者获胜 "规则会产生一个因果+ 而非 RTC 一致的系统，而 "全部退回 "冲突则会产生一个因果+ 而非 RTC 一致的系统。处理程序将提供这两种属性。

可线性化系统。线性化可通过单个提交点（如主拷贝系统 [4, 39]，可通过两阶段提交协议*急切*复制数据 [44]）或分布式协议（如 Paxos [33]）来实现。法定人数系统不是到处复制内容，而是确保读写集重叠以实现线性化[22, 25]。

如前所述，CAP 规定可线性化系统的延迟不能低于其数据中心间往返延迟；只有在最近才被用于广域操作，而且只有在可以牺牲 ALPS 的低延迟时才会使用 [9]。CRAQ [48]可以在本地区域完成读取，但在其他情况下需要广域操作以确保线性化。

事务与大多数文件系统或键值存储不同，数据库界长期以来一直在考虑通过使用读写事务来实现多个键*的*一致性。在许多商业数据库系统中，单个主数据库会跨键执行事务，然后将其事务日志*懒散地*发送到其他副本（可能是广域副本）。通常情况下，这些异步副本是只读的，与 COPS 的任意写副本不同。当今的大型数据库通常会将数据分区（或分片）到多个数据库实例上 [17, 38, 42]，这与一致散列中的情况非常相似。转换只在单个分区内进行，而 COPS 可以跨节点/分区建立因果关系。

一些数据库系统支持跨分区和/或数据中心（在数据库文献中这两者都被视为独立*站点*）的事务处理。例如，R* 数据库 [37] 使用进程树和两阶段锁定来实现多站点事务处理。然而，这种两阶段锁定使系统无法保证可用性、低延迟或分区容错。Sinfo- nia [1] 通过轻量级两阶段提交协议为分布式共享内存提供了 "迷你 "事务，但只考虑了单个数据中心内的操作。最后，最近推出的广域键值存储 Walter [47] 提供了跨键值的事务一致性（包括写入，这与 COPS 不同），并包括在某些情况下允许在单个站点内执行事务的操作。不过，COPS 注重可用性和低延迟，而 Walter 则强调事务保证：确保键之间的因果关系可能需要在广域内进行两阶段提交。此外，在 COPS 中，可扩展的数据中心是首要设计目标，而 Walter 的站点目前由单机组成（作为事务的单一序列化点）。

8. 结论

当今的大规模广域系统可为客户端提供 "始终在线 "的低延迟操作，但代价是一致性保障薄弱和应用逻辑复杂。本文介绍的 COPS 是一种可扩展的分布式存储系统，可在不牺牲 ALPS 特性的情况下提供因果+ 一致性。COPS 通过跟踪和显式检查每个集群中的写入之前是否满足因果依赖关系来实现因果+一致性。COPS-GT 以 COPS 为基础，引入了获取事务，使客户端能够获得多个密钥的一致性视图；COPS-GT 还进行了优化，以减少状态、最小化多轮协议并降低复制开销。我们的评估表明，COPS 和 COPS-GT 可提供低延迟、高吞吐量和可扩展性。

致谢。我们要特别感谢 SOSP 项目委员会和我们的指导者 Mike Dahlin，感谢他们广泛的意见和 Mike 周到的互动，这些意见和互动极大地改进了这项工作的表述方式，实际上也改进了我们自己对这项工作的看法。杰夫-特雷斯（Jeff Terrace）、埃里克-诺德斯特罗姆（Erik Nordstro¨m）和戴维-舒（David Shue）对这项工作提出了有用的意见；维杰-瓦苏德凡（Vijay Vasudevan）在FAWN-KV方面提供了有益的帮助；萨潘-巴蒂亚（Sapan Bhatia）和安迪-巴维尔（Andy Bavier）帮助我们在VICCI测试平台上进行实验。这项工作得到了美国国家科学基金会（NSF）的资助（CAREER CSR-0953197 和 CCF-0964474）、VICCI（NSF
MRI-1040123奖）、谷歌的捐赠以及英特尔云计算科技中心。

参考文献

[1] M.K. Aguilera、 A. Merchant、 M. Shah、 A. Veitch 和 C. Karamanolis。Sinfonia：构建可扩展分布式系统的新范式。*ACM TOCS*, 27(3), 2009.

[2] M.Ahamad, G. Neiger, P. Kohli, J. Burns, and P. Hutto.因果记忆：定义、实现与编程。*分布式 计算》*，9(1)，1995 年。

[3] M.Al-Fares, A. Loukissas, and A. Vahdat.可扩展的商品数据 中心网络架构。在 *SIGCOMM*，2008 年 8 月。

[4] P.Alsberg and J. Day.分布式 资源弹性共享原则。In *Conf.软件工程*，1976 年 10 月。

[5] D.G. Andersen、J. Franklin、M. Kaminsky、A. Phanishayee、L. Tan 和 V. Vasudevan。FAWN：懦弱节点的快速阵列。在 *SOSP*，2009 年 10 月 。

[6] T.E. Anderson、M. D. Dahlin、J. M. Neefe、D. A. Patterson、D. S. Roselli 和 R. Y. Wang。Wang.无服务器网络文件系统。*ACM TOCS*, 14(1), 1996.

[7] Apache Thrift。http://thrift.apache.org/，2011 年。

[8] H.H. Attiya 和 J. L. Welch.顺序一致性与线性化。*ACM TOCS*, 12(2), 1994.

[9] J.Baker、 C. Bond、 J. C. Corbett、 J. Furman、 A. Khorlin、 J. Larson、 J .-M.Leon, Y. Li, A. Lloyd, and V. Yushprakh.Megastore：为交互式服务提供可扩展的高可用存储。2011年1月，*CIDR*，。

[10] N.Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani、P.Yalagandula 和 J. Zheng。PRACTI 复制。2006 年 5 月，*NSDI* 。

[11] K.P. Birman 和 T. Joseph。在分布式 系统中利用虚拟同步。在 *SOSP*，1987 年 11 月。

[12] K.P. Birman 和 R. V. Renesse.*使用 ISIS 工具包的可靠分布式计算* 。IEEE Comp. Soc. Press, 1994.Soc. Press, 1994.

[13] E.Brewer.走向稳健的分布式系统。PODC 主题演讲，2000 年 7 月 。

[14] D.R. Cheriton 和 D. Skeen.理解因果 和完全有序通信的局限性。在 *SOSP*，1993 年 12 月。

[15] B.B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bo- hannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni.Jacobsen, N. Puz, D. Weaver, and R. Yerneni.PNUTS：雅虎的托管数据服务平台。在 *VLDB*，2008 年 8 月。

[16] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman、A.Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels.Dynamo：亚马逊的高可用键值存储。2007 年 10 月，*SOSP*。

[17] D.DeWitt、S. Ghandeharizadeh、D. Schneider、A. Bricker、H.-I. Hsiao 和 R. Rasmussen。Hsiao, and R. Rasmussen.伽马数据库机项目。*知识 与数据工程*，2（1），1990 年。

[18] fawn-kv. `https://github.com/vrv/FAWN-KV`, 2011.

[19] B.Fitzpatrick.Memcached: a distributed memory object caching system. `http://memcached.org/`, 2011.

[20] A.Fox, S. D. Gribble, Y. Chawathe, E. A. Brewer, and P. Gauthier.基于集群的可扩展网络服务。在 *SOSP*，1997 年 10 月。

[21] S.Ghemawat, H. Gobioff, and S.-T. Leung.Leung.谷歌文件系统在 *SOSP*, 2003 年 10 月。

[22] D.K. Gifford.复制数据的加权投票。在 *SOSP*，1979 年 12 月。

[23] S.Gilbert 和 N. Lynch.布鲁尔猜想与一致、可用、分区容忍网络服务的可行性。*ACM SIGACT News*, 33(2), 2002.

[24] A.格林伯格、J. R. 汉密尔顿、N. 詹恩、S. 坎杜拉、C. 金、P. 拉希里、
D.D. A. Maltz、P. Patel 和 S. Sengupta。VL2：可扩展且灵活的数据中心网络。在 *SIGCOMM*，2009 年 8 月。

[25] M.Herlihy.抽象数据 类型的法定人数共识复制方法。*ACM TOCS*，4(1)，1986 年 2 月。

[26] M.P. Herlihy 和 J. M. Wing.线性化：并发对象的正确性条件 .*ACM TOPLAS*，12（3），1990.

[27] D.Karger, E. Lehman, F. Leighton, M. Levine, D. Lewin, and R. Pan- i g r a h y .一致散列和随机树：缓解万维网热点的分布式缓存协议。在 *STOC*, 1997 年 5 月。

[28] J.Kistler 和 M. Satyanarayanan.Coda 文件系统中的断开操作。*ACM TOCS*，10（3），1992 年 2 月。

[29] R.Ladin、B. Liskov、L. Shrira 和 S. Ghemawat。使用懒复制提供高 可用性。*ACM TOCS*，10（4），1992。

[30] A.Lakshman 和 P. Malik。Cassandra - 一种去中心化的结构化 存储系统。在 *LADIS*，2009 年 10 月。

[31] L.Lamport.分布式 系统中的时间、时钟和事件排序。*通讯。ACM*，21（7），1978.

[32] L.Lamport.如何制造能正确 执行多进程程序的多处理器计算机。*IEEE Trans.计算机》*，28（9），1979 年。

[33] L.Lamport.The part-time parliament.*ACM TOCS*, 16(2), 1998.

[34] R.J. Lipton 和 J. S. Sandberg.PRAM：可扩展的共享内存。技术报告 TR-180-88，普林斯顿大学计算机科学系，1988 年。Sci.

[35] P.Mahajan, L. Alvisi, and M. Dahlin.一致性、可用性和收敛性。技术报告 TR-11-22，德克萨斯大学奥斯汀分校， Comp.Sci.

[36] J.Misra.异步硬件系统中的内存访问公理 tems.*ACM TOPLAS*，8（1），1986 年 1 月。

[37] C.Mohan, B. Lindsay, and R. Obermarck.R* 分布式数据库管理系统中的事务管理。*ACM Trans.数据库 Sys.*，11（4），1986 年。

[38] MySQL。http://www.mysql.com/，2011 年。

[39] B.M. Oki 和 B. H. Liskov.视图标记复制：通用 主副本。见 *PODC*，1988 年 8 月。

[40] D.D. S. Parker、G. J. Popek、G. Rudisin、A. Stoughton、B. J. Walker、
E.Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline.分布式系统中相互不一致的检测。*IEEE Trans.软件 Eng.*, 9(3), 1983.

[41] K.Petersen, M. Spreitzer, D. Terry, M. Theimer, and A. Demers.弱一致性复制的灵活更新传播。*SOSP*, 1997 年 10 月 .

[42] PostgresSQL。http://www.postgresql.org/，2011 年。

[43] 伏地魔项目。http://project-voldemort.com/，2011 年。

[44] D.Skeen.分布式系统崩溃恢复的形式化模型
IEEE Trans.软件工程》，9（3），1983 年 5 月。

[45] Snappy. `http://code.google.com/p/snappy/`, 2011.

[46] J.索贝尔扩大规模。Facebook 工程学博客，2008 年 8 月 20 日。

[47] Y.Sovran、R. Power、M. K. Aguilera 和 J. Li。地理复制系统的事务存储 。在 *SOSP*，2011 年 10 月。

[48] J.Terrace and M. J. Freedman.CRAQ 上的对象存储：针对多读工作负载的高吞吐量链式复制。*USENIX ATC*，2009 年 6 月。

[49] D.B. Terry、A. J. Demers、K. Petersen、M. Spreitzer、M. Theimer 和
B.W. Welch.弱一致性复制数据的会话保证。In *Conf.Parallel Distributed Info.Sys.*, Sept.

[50] R.H. Thomas.多副本数据库并发控制的多数共识方法 .*ACM Trans.数据库系统》*，4（2），1979 年。

[51] R. van Renesse 和 F. B. Schneider。支持 高吞吐量和可用性的链式复制。在 *OSDI*，2004 年 12 月。

[52] `VICCI. http://vicci.org/`, 2011.

[53] H.Yu 和 A. Vahdat.复制服务连续一致性模型的设计与评估。在 *OSDI*，2000 年 10 月。

a.因果关系的　　正式定义+

我们首先为一个只有获取和放入操作（读取和写入）的系统介绍了具有收敛冲突处理（因果+ 一致性）的因果一致性，然后引入获取事务。我们使用

该模型主要源自 Ahamad 等人[2]的研究，而 Ahamad 等人的研究又源自 Herlihy 和 Wing [26] 以及 Misra [36]的研究。

最初的因果一致性模型[2]，术语经过修改以符合本文的定义：

系统是一组有限的执行线程（也称为线程），它们通过键值存储进行交互，键值存储由一组有限的键组成。让 $T = \{t_1, t_2, ., t_n\}$ 是线程集。的局部历史记录 L_i 一线是一系列 get 和 put 操作。如果操作 σ_1 在 L_i 的 σ_2 之前，我们写成 $\sigma_1 \rightarrow_i \sigma_2$。历史 $H = (L_1, L_2, \ldots, L_n)$ 是所有执行线程的本地历史记录集合。A

H 的序列化 S 是 H 中所有操作的线性序列，其中键上的每个 get 都返回其最近的前一个 put（如果不存在前一个 put，则返回±）。如果对于 S 中的任何操作 σ_1 和 σ_2，$\sigma_1 \rightarrow \sigma_2$ 意味着 S 中的 σ_1 先于 σ_2，则序列化 S 尊重顺序 →。

由于一个键的值可能有多个put，因此可能有不止一个put-into 令。[12]H 上的puts-into 顺序'→ 是具有以下属性的任何关系：

- 如果 σ_1 '→ σ_2，那么存在一个键 k 和值 v，使得操作 σ_1 := put(k,v)和 σ_2 := get(k)=v。
- 对于任何操作 σ_2，至多存在一个 σ_1，对于该操作 σ_1 '→ σ。[2]
- 如果对某个 k,v 来说，σ_2 := get(k)=v，并且不存在 σ_1，使得 σ_1 '→ σ_2，那么 v = ±。也就是说，前面没有 put 的 get 必须取回初始值。

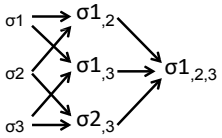
[12]如果且

只有当下列条件之一成立时

- $\sigma_1 \rightarrow \sigma_2$ for some t_i (在 L_i 中，σ_1 在 σ_2 之前)；
- σ_1 '→ σ_2 (σ_2，获得 σ_1 的值)；或
- 还有一些其他操作 σ^J，使得 $\sigma_1 \sim \sigma^J \sim \sigma$。[2]

加入趋同冲突处理。对同一密钥的两个操作，即 σ_1 :=put(k,v_1) 和 σ_2 :=put(k,v_2)，在下列情况下*发生冲突*它们之间没有因果关系：σ_1 /~ σ_2 和 σ_2 /~ σ。[1]

*收敛式冲突处理函数*是一种关联的、组合的*冲突处理函数。*突变函数，对键进行一系列冲突操作，最终产生该键的一个（可能是新的）最终值。该函数必须产生相同的最终值，与观察冲突更新的顺序无关。这样，一旦每个副本都观察到了



处理程序函数的交换性和关联性使

因此，无论顺序如何，最终输出都是相同的。此外，它将在所有原始冲突写入以及处理程序函数应用所产生的任何中间值之后进行因果排序。如果处理程序观察到多对产生相同输出值的冲突更新（例如上图中的最终输出），它必须只输出一个值，而不是

同一值的多个实例。

为了防止客户端看到多个相互冲突的值并进行推理，我们限制每个**客户端**线程的投入集为无冲突集，表示为 p_{cf_i}。一个投入集是*无冲突的*

如果 σ∀σ_j, σ_k e p_{cf_i}, σ_j 和 σ_k 不冲突；也就是说，它们要么是或因果相关的键。对于

例如，在三个冲突的投放示例中，p_{cf_i} 可能包括 σ_1, $\sigma_{1,2}$, 和 $\sigma_{1,2,3}$, 但不包括 σ, σ, σ_{2}31,3, 或 $\sigma_{2,3}$。无冲突属性适用于客户端线程，而不是处理线程。处理程序线程必须能够从冲突的 puts 中获取值，以便推理和解决这些冲突；客户端线程不应看到冲突，这样它们就不必推理这些冲突。

添加处理程序线程可以模拟冲突处理所提供的新功能。限制投入集加强了从因果到因果+的一致性。存在不是因果+的因果执行：例如，如果 σ_1 和 σ_2 发生冲突，在一个因果但不是因果+的系统中，客户端可能会获得由 σ_1 设置的值，然后再获得由 σ_2 设置的值。另一方面，不存在不是因果+的因果+执行，因为因果+只是对因果一致性引入了额外的限制（更小的投放集）。

如果 H 是一段历史，t_i 是一个线程，　包括所有业务则让 A^H_{i+pcfi}

t 的局部历史中$_i$，以及 H 中无冲突的投入集 p_{cf_i}。如果历史 H 有一个因果顺序 ~，使得

因果+：对于执行 t_i 的每个*客户端*线程，都有一个串行的 A^H_{i+pcfi} 化 S_i 尊重 ~。

密钥的冲突更新，它们就会独立地就相同的最终值达成一致。

我们将收敛冲突处理建模为一组有别于普通客户端线程的*处理*线程。处理程序对一对冲突值 (v_1, v_2) 进行操作，产生一个新值 $newval = h(v_1, v_2)$。根据交换性，$h(v_1, v_2) = h(v_2, v_1)$。为了产生新值，处理线程必须先读取 v_1 和 v_2，然后再放入新值，因此 *newval* 在因果关系上排序在两个原始值之后：v_1 ～ *newval* 和 v_2 ～ *newval*。

如果有两个以上的冲突更新，就会有多个处理程序线程被调用。对于三个值，有几种可能的顺序来解决成对的冲突更新：

[12] COPS 系统用版本号唯一标识值，因此只有一个投入顺序，但一般因果+一致性并不一定如此。

如果数据存储只允许因果+历史，那么它就是因果+*一致的*。

引入获取事务。为了在模型中加入获取事务，我们重新定义了 puts-into 顺序，使其与 N 个值的每个获取事务关联 N 个 put 操作 put(k,v)、get-trans($[k_1, \ldots, k_N]$)=$[v_1, \ldots, v_N]$。现在，'→'投入阶是具有以下性质的任何关系：

- 如果 $\sigma_1 \text{'}\rightarrow \sigma_2$，则存在 k 和 v，使得 σ_1 := put(k,v)和 σ_2 := get trans([. . . ,k, . .])=[. . . ,v, . .]。也就是说，对于 get 事务的每个组成部分，都存在一个前面的放。
- 对于获取交易的每个组件 σ_2，最多存在一个 σ_1，其 $\sigma_1 \text{'}\rightarrow \sigma_2$。
- 如果 σ_2 := get trans([. . . ,k, . .])=[. . . ,v, . .] 对于某个 k,v 而不存在 σ_1，使得 $\sigma_1 \text{'}\rightarrow \sigma_2$，那么 v=±。也就是说，前面没有 put 的 get 必须取回初始值。