我们为分布式机器学习问题提出了参数服务器框架。数据和工作负载分发到各工作节点上，而服务器节点维护全局共享参数，这些参数以密集或稀疏向量和矩阵表示。该框架管理节点之间的异步数据通信，并支持灵活的一致性模型，弹性可伸缩性和连续容错。

为了论证所提出的框架的可扩展性，在解决稀疏逻辑回归到潜在狄利克雷分配和分布式草图这些问题时，我们用数十亿个实例和参数来显示 PB 级实际数据的实验结果。

1 介绍

分布式优化和论证正在成为解决大规模机器学习问题的先决条件。 在规模上，由于数据的增长和由此产生的模型复杂性，任何一台机器都不能快速解决这些问题，而这往往体现在参数数量上的增加。 然而，实现高效的分布式算法并不容易。 密集的计算工作量和数据通信量都需要仔细的系统设计。

实际数量的训练数据可以在 1TB 到 1PB 之间变化。 这使得人们可以创建 10^9 到 10^12 个参数的强大而复杂的模型。 这些模型通常由所有工作节点全局共享，这些工作节点在执行计算以改进它时必须经常访问共享参数。共享带来三个挑战：

1）访问参数需要大量的网络带宽。

2）许多机器学习算法是顺序的。当同步成本和机器延迟较高时，由此产生的障碍会影响性能。

3）在规模上，容错是至关重要的。学习任务通常在云环境中执行，这种环境下，机器可能不可靠，可能会抢占作业。

$\approx$ #machine $\times$ time	# of jobs	failure rate
100 hours	13,187	7.8%
1,000 hours	1,366	13.7%
10,000 hours	77	24.7%

Table 1: Statistics of machine learning jobs for a three month period in a data center.

为了说明最后一点，我们从一家大型互联网公司的一个集群中收集了三个月期间的所有工作日志。 我们在表 1 中显示了为生产环境提供服务的批量机器学习任务的统计数据。这里，任务失败主要是由于在缺失必要的容错机制的情况下被抢占或丢失机器资源所致。

在实际部署中容错性是必需的，不像在研究的环境中，作业完全在群集上运行而没有争夺资源的状况。

1.1 贡献

自推出以来，参数服务器框架在学术界和工业界迅速普及。本文描述了参数服务器的第三代开源实现，它着重于分布式推理的系统方面。它为开发人员提供了两个优势：首先，通过分解出机器学习系统中常用的组件，它使特定于应用程序的代码保持简洁。同时，作为针对系

统级优化的共享平台，它提供了一个强大、多功能和高性能的实现，能够处理从稀疏逻辑回归到主题模型和分布式草图的各种算法。我们的设计决策根据实际系统中工作负载的情况进行改变。我们的参数服务器提供了五个关键特性：

高效通信：异步通信模型不会阻塞计算（除非需要）。它针对机器学习任务进行了优化，以减少网络流量和开销。

灵活的一致性模型：宽松的一致性进一步隐藏了同步成本和延迟。我们允许算法设计者平衡算法收敛速度和系统效率。最好的折衷取决于数据，算法和硬件。

弹性可伸缩性：可以在不重新启动运行框架的情况下添加新节点。

容错性和耐久性：在 1 秒内恢复并修复非灾难性机器故障，而不中断计算。矢量时钟确保网络分区和故障前所定义好的行为不受影响。

易用性：全局共享参数表示为（潜在稀疏）矢量和矩阵，以促进机器学习应用程序的开发。 线性代数数据类型带有高性能多线程库。

所提出系统的新颖之处在于实现功能上的协同效应，选择正确的系统技术，使适应机器学习算法以及修改机器学习算法以使系统更友好。尤其，我们可以放宽一些其他有难度的系统约束，因为相关的机器学习算法相当抗干扰。结果是第一个能够扩展到工业规模的通用 ML 系统。

1.2 工程挑战

在解决分布式数据分析问题时，读取和更新不同工作节点间共享参数的问题无处不在。参数服务器框架为工作节点之间的模型参数和统计信息之间的聚合和同步提供了一种有效的机制。每个参数服务器节点仅维护一部分参数，并且每个工作节点在运行时通常只需要这些参数的子集。在构建高性能参数服务器系统时出现两个关键挑战：

通讯。虽然参数可以更新为传统数据存储区中的键值对，但仅仅使用此抽象方法效率不高：值通常较小（浮点或整数），并且将每次更新作为键值操作发送的开销很高。

我们对改善这种情况的见解来自许多学习算法将参数表示为结构化数学对象（如矢量，矩阵或张量）的观察结果。在每个逻辑时间（或迭代）中，通常会更新对象的一部分。也就是说，工作节点通常会发送矢量的一部分，或矩阵的整行。这提供了一个机会，可以自动批量处理参数服务器上的更新及其处理，并且可以高效地执行一致性跟踪。

容错。如前所述，在规模上是至关重要的，并且为了高效运行，它不一定需要完全重启长时间运行的计算。在服务器之间实时复制参数支持热故障转移。故障转移和自我修复分别通过将机器移除或添加作为故障或修理依次支持动态缩放。

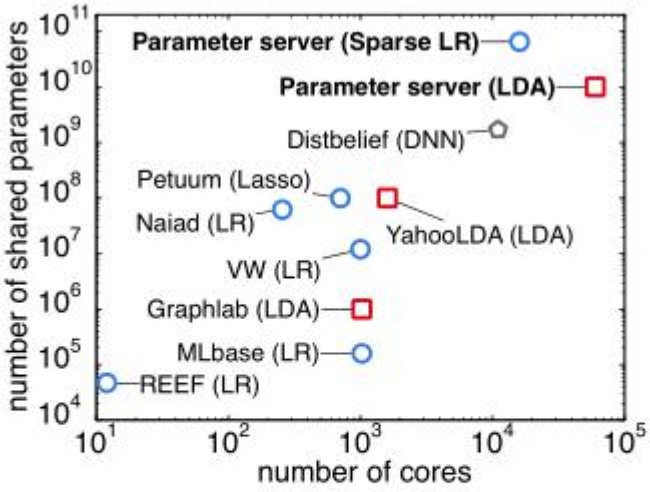

Figure 1: Comparison of the public largest machine learning experiments each system performed. Problems are color-coded as follows: Blue circles — sparse logistic regression; red squares — latent variable graphical models; grey pentagons — deep networks.

	Shared Data	Consistency	Fault Tolerance
Graphlab [34]	graph	eventual	checkpoint
Petuum [12]	hash table	delay bound	none
REEF [10]	array	BSP	checkpoint
Naiad [37]	(key,value)	multiple	checkpoint
Mlbase [29]	table	BSP	RDD
Parameter Server	(sparse) vector/matrix	various	continuous

Table 2: Attributes of distributed data analysis systems.

Figure 1 概述了在多个系统上执行的最大规模监督和无监督机器学习实验。在可能的情况下，我们与每个系统的作者（截至 2014 年 4 月的数据）确认了扩展的限制。显而易见，我们能够比任何其他已发布的系统在数量级上覆盖更多数量级的数据。此外，Table 2 概述了几种机器学习系统的主要特性。我们的参数服务器在一致性方面提供最大程度的灵活性。它是唯一提供连续容错功能的系统。其原生数据类型使其对数据分析特别友好。

1.3 相关工作

亚马逊，百度，Facebook，谷歌，微软和雅虎已经实现了相关系统。开源代码也存在，例如 YahooLDA 和 Petuum。此外，Graphlab 支持尽力而为模型的参数同步。

引入的第一代这种参数服务器缺乏灵活性和性能，它将 memcached 分布式（密钥，值）存储重新用作同步机制。YahooLDA 通过使用用户可定义的更新原语（set，get，update）和更具原则性的负载分配算法实现专用服务器，从而改进了此设计。第二代应用程序特定的参数

服务器也可以在 Distbelief 和同步机制中找到。Petuum 开展了一个通用平台的第一步。它使用有界延迟模型改进了 YahooLDA，同时对工作线程模型进一步加以限制。我们描述克服这些限制的为第三代系统。

最后，将参数服务器与用于机器学习的更通用的分布式系统进行比较是有用的。其中有几个要求同步迭代通信。它们可以很好地扩展到数十个节点，但在大规模情况下，随着节点运行缓慢越来越明显时，这种同步会带来挑战。基于 Hadoop 的 Mahout 和基于 Spark 的 MLI，都采用迭代 MapReduce 框架。Spark 和 MLI 的一个关键点是在迭代之间保持状态，这是参数服务器的核心目标。

分布式 GraphLab 改为使用图抽象异步调度通信。目前，GraphLab 缺乏基于 map / reduce 的框架的弹性可伸缩性，并且它依赖于粗粒度快照进行恢复，这两者都阻碍了可伸缩性。它对某些算法的适用性受限于它缺乏全局变量同步作为一个有效的一级原语。从某种意义上说，参数服务器框架的核心目标是吸取 GraphLab 异步的好处，而不受其结构限制。

Piccolo 使用与参数服务器相关的策略来共享和聚集机器之间的状态。其中，workres 在本地预先聚集状态并将更新传输到保持聚集状态的服务器。因此，它大部分实现了我们系统功能的一个子集，缺乏机制学习规范的优化：消息压缩，复制和通过依赖关系图表示的变量一致性模型。

2 机器学习

机器学习系统广泛用于网络搜索，垃圾邮件检测，推荐系统，计算广告和文档分析。这些系统自动从示例（训练数据）中学习模型，通常包含三个组件：特征提取，目标函数和学习。

特征提取处理原始训练数据，例如文档，图像和用户查询日志，以获得特征向量，其中每个特征都捕获训练数据的属性。预处理可以通过现有的框架（如 MapReduce）高效地执行，这超出了本文的范围。

2.1 目标

许多机器学习算法的目标可以通过一个“目标函数”来表达。这个函数捕捉学习模型的属性，例如在将电子邮件分类为正常邮件和垃圾邮件的情况下出现的错误较低，在估计文档中的主题时数据解释得如何，或者在绘制数据的情况下对统计进行简要总结。

学习算法通常将这个目标函数最小化以获得该模型。一般来说，没有封闭的解决方案;相反，学习从最初的模型开始。它通过处理训练数据，可能多次迭代地改进这个模型，以接近解决方案。当发现（近似）最优解或者认为模型被收敛时，它停止。

训练数据可能非常大。例如，一家大型互联网公司使用一年广告印象日志来训练广告点击预测，这将拥有数万亿训练范例。每个训练样例通常表示为可能非常高维的“特征向量”。因此，训练数据可能包含万亿长度的特征向量。迭代处理这种大规模数据需要巨大的计算和带宽资源。此外，数十亿的新广告印象可能每天都会到达。将此数据添加到系统中通常会提高预测准确度和覆盖率。但它也需要学习算法每天运行，有时可能需要实时。这些算法的有效执行是本文的重点。

为了完善我们系统中的设计，接下来我们简要概述两种广泛使用的机器学习技术，我们将用它们来演示参数服务器的功效。

2.2 风险最小化

机器学习问题最直观的变体是风险最小化。“风险”大体上是预测误差的一种度量。例如，如果我们要预测明天的股票价格，风险可能是预测与股票实际价值之间的偏差。

训练数据由 n 个示例组成。xi 是第 i 个这样的例子，并且通常是长度为 d 的向量。如前所述，n 和 d 分别可能在数十亿到数万亿的样例和维度上。在很多情况下，每个训练样例 xi 都与一个标签 yi 相关联。例如，在广告点击预测中，yi 对于“点击”可能是 1，对于“未点击”可能是-1。

风险最小化学习出模型，该模型可以预测未来示例 x 相对应的值 y。该模型由参数 w 组成。在最简单的示例中，模型参数可能是广告展示中每个功能的“点击率”。 为了预测是否会点击新的印象，系统可以根据印象中呈现的特征简单地总结其“点击率”，即

$$x^T w := \sum_{j=1}^{d} x_j w_j$$

然后根据符号来决定。

在任何学习算法中，训练数据量与模型大小之间存在重要关系。一个更详细的模型通常会提高准确性，但会达到一个点：如果训练数据太少，高度详细的模型就会过度适应，并且变成仅仅是一个系统，它可以独特地记录出训练集中的每个数据项。另一方面，一个太小的模型将无法捕捉对做出正确决策至关重要的数据的重要和相关属性。

正则化风险最小化是寻找平衡模型复杂性和训练误差的模型的一种方法。它通过最小化两个项的和来实现：表示训练数据上的预测误差的损失 L(x,y,w)和惩罚模型复杂性的正则化器 Ω[w]。一个好的模型是低错误和低复杂度的模型。所以我们努力最小化

$$F(w) = \sum_{i=1}^{n} l(x_i, y_i, w) + \Omega(w)$$

所使用的具体损失和正则化函数对于机器学习算法的预测性能是重要的，但对于本文的目的而言相对不重要：我们提出的算法可以用于所有最常用的损失函数和正则化器。

在 5.1 节我们使用高性能的分布式学习算法来评估参数服务器。为了简单起见，我们描述了一个更简单的模型，称为分布式次梯度下降。

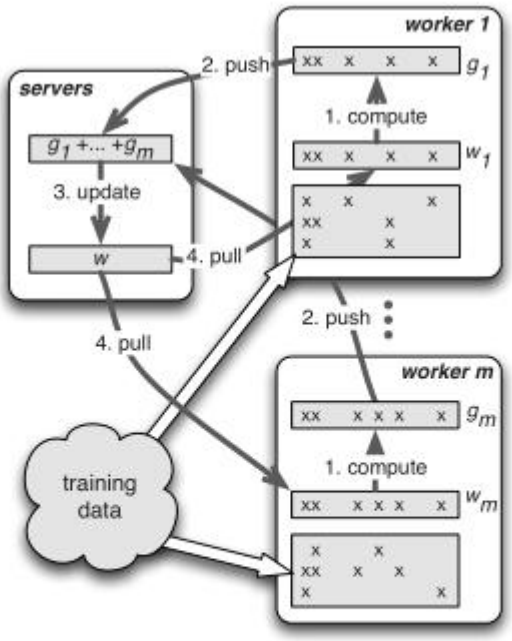

Figure 2: Steps required in performing distributed subgradient descent, as described e.g. in [46]. Each worker only caches the working set of w rather than all parameters.

Algorithm 1 Distributed Subgradient Descent

Task Scheduler:

issue LoadData() to all workers
for iteration $t = 0, \ldots, T$ **do**
    issue WORKERITERATE(t) to all workers.
end for

Worker $r = 1, \ldots, m$:

function LOADDATA()
    load a part of training data $\{y_{i_k}, x_{i_k}\}_{k=1}^{n_r}$
    pull the working set $w_r^{(0)}$ from servers
end function
function WORKERITERATE(t)
    gradient $g_r^{(t)} \leftarrow \sum_{k=1}^{n_r} \partial \ell(x_{i_k}, y_{i_k}, w_r^{(t)})$
    push $g_r^{(t)}$ to servers
    pull $w_r^{(t+1)}$ from servers
end function

Servers:

function SERVERITERATE(t)
    aggregate $g^{(t)} \leftarrow \sum_{r=1}^{m} g_r^{(t)}$
    $w^{(t+1)} \leftarrow w^{(t)} - \eta \left(g^{(t)} + \partial \Omega(w^{(t)}) \right)$
end function

如 Figure 2 和算法 1 所示，训练数据分割到所有工作节点上，这些节点共同学习出参数向量 w。该算法迭代运行。在每次迭代中，每个节点独立地使用自己的训练数据来确定应该对 w 做些什么改变，以便接近最优值。由于每个节点的更新仅反映其自己那部分的训练数据，因此系统需要一种机制来允许这些更新进行整合。它通过将这些更新表达为子梯度（参数向量 w 应该移位的方向）并在将所有子梯度应用于 w 之前整合起来。通过在算法设计中考虑正确的学习速率 η，这些梯度通常会规模化缩小，以便确保算法快速收敛。

算法 1 中最花费时间的步骤是计算子梯度以更新 w。这项任务分配给 worker，每个 worker 都执行 WORKERITERATE。作为其中的一部分，worker 计算 $w^T x_{ik}$，这对于高维 w 是不可行的。幸运的是，当且仅当其中一些训练数据用于参考输入时，worker 需要知道 w 的值。

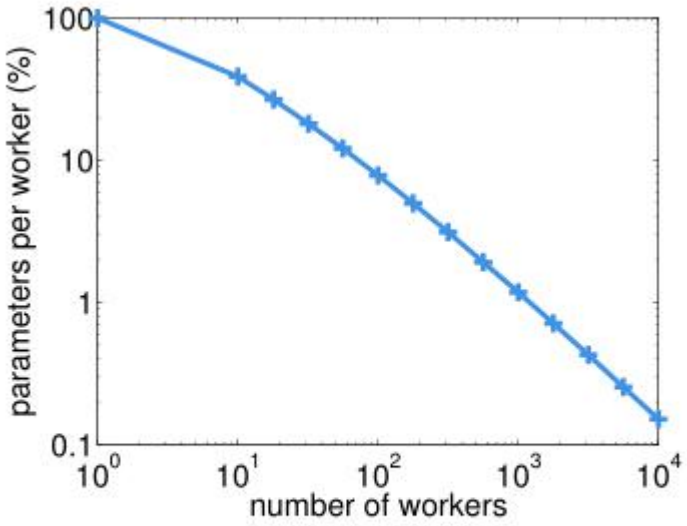

Figure 3: Each worker's set of parameters shrinks as more workers are used, requiring less memory per machine.

例如，在广告点击预测中，关键特征之一是广告中的单词。如果只有极少数广告包含 OSDI 2014 短语，那么大多数 worker 不会对 w 中的相应条目产生任何更新，因此不需要此条目。尽管 w 的总大小可能会超过单个机器的容量，但特定 worker 所需的条目工作集可以在本地缓存。为了说明这一点，我们将数据随机分配给 worker，然后对第 5.1 节中使用的数据集上每个 worker 的平均工作集大小进行计数。Figure 3 显示，对于 100 个 worker，每个 worker 只需要总参数的 7.8％。有 10,000 个 worker，这降低到 0.15％。

2.3 生成模型

在第二类主要的机器学习算法中，要应用于训练样例的标签是未知的。这样的设置称为无监督算法（对于标记的训练数据，可以使用监督或半监督算法）。它们试图捕获数据集的基础结构。例如，这个领域的一个常见问题是主题建模：给定文档集合，推断每个文档中包含的主题。

当运行时，例如 SOSP'13 proceedings，算法可能会产生诸如“分布式系统”，“机器学习”和“性能”等主题。算法从文档本身的内容推断出这些主题，而不是从外部主题列表中获取。在推荐系统的内容个性化等实际环境中，这些问题的规模是巨大的：数以亿计的用户和数十

亿的文档，这对于跨大型集群并行化算法至关重要。

由于它们的规模和数据量，这些算法在引入第一代参数服务器之后才在商业上适用。主题模型的一个关键挑战是必须共享参数，这些参数描述文件如何生成当前的估计。

流行的主题建模方法是潜狄利克雷分配（LDA）。虽然统计模型是完全不同的，但所得到的学习算法与算法 1 非常相似。然而，关键的区别在于更新步骤不是一个梯度计算，而是模型如何解释对一个文档的评估。此计算需要访问每次访问文档时更新的辅助元数据。由于文档的数量，元数据通常会被读取并当文档处理完后写回到磁盘。

该辅助数据是分配给文档的每个单词的主题集合，并且正在学习的参数 w 由单词出现的相对频率组成。

与以前一样，每个 worker 只需要存储处理文档中出现的单词的参数。因此，在 worker 之间分发文档的效果与前一节中的相同：我们可以处理比单个 worker 可能拥有的更大的模型。

3 架构

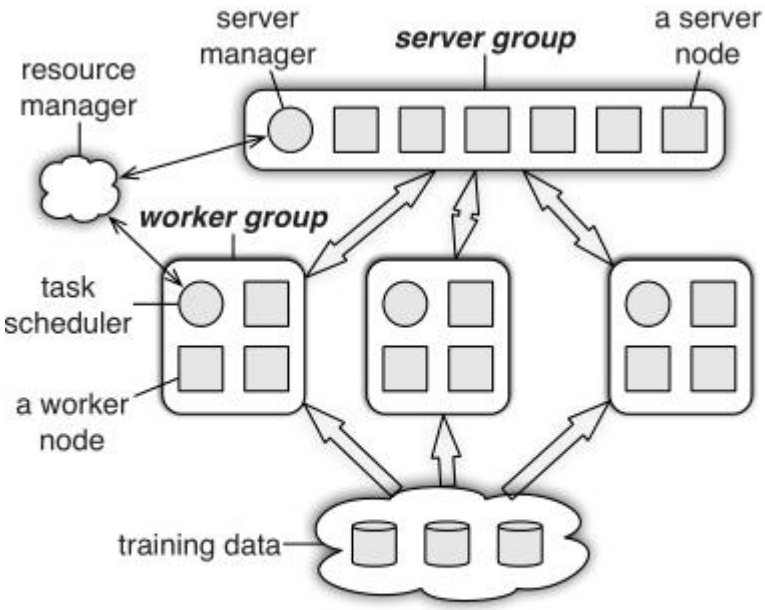

Figure 4: Architecture of a parameter server communicating with several groups of workers.

参数服务器的一个实例可以同时运行多个算法。参数服务器节点分组为一个服务器组和多个工作组，如 Figure 4 所示。服务器组中的一个服务器节点维护全局共享参数的一个分区。服务器节点彼此通信以复制和/或迁移用于可靠性和规模变化的参数。服务器管理器节点维护服务器元数据的一致视图，例如节点活跃性和参数分区的签名。

每个工作组运行一个应用程序。节点 worker 通常在本地存储一部分训练数据以计算当地的统计数据，如梯度。worker 只与服务器节点通信，更新和检索共享参数。每个工作组都有

一个调度程序节点。它将任务分配给 worker 并监视其进度。如果 worker 被添加或删除，它会重新规划未完成的任务。

参数服务器支持独立的参数名称空间。这允许工作组将其共享的参数集与其他组分离开。几个工作组也可能共享相同的名称空间：我们可能使用多个工作组来解决相同的深度学习应用程序以增加并行化。另一个例子是一些模型正在被一些节点频繁查询，例如使用这种模型的在线服务。同时，随着新的训练数据到达，模型由不同组的工作节点更新。

参数服务器旨在简化开发分布式机器学习应用程序，例如第 2 节中讨论的那些应用程序。共享参数以(key,value)向量的形式表示，以促进线性代数运算（第 3.1 节）。它们分布在一组服务器节点上（第 4.3 节）。任何节点都可以推出它的本地参数并从远程节点中提取参数（第 3.2 节）。默认情况下，工作负载或任务由工作节点执行;然而，它们也可以通过用户定义的功能分配给服务器节点（第 3.3 节）。任务是异步的并且并行运行（3.4 节）。参数服务器为算法设计者提供了通过任务依赖关系图（第 3.5 节）选择一致性模型以及传送参数子集（第 3.6 节）。

3.1 (key,value)向量

节点之间共享的模型可以表示为一组(key，value)键值对。例如，在损失最小化问题中，这个对是一个特征 ID 及其权重。对于 LDA，这一对是词汇 ID 和主题 ID 以及计数的组合。模型的每个条目都可以在本地读取或写入，也可以通过 key 值远程读取。这种(key，value)抽象被现有方法广泛采用。

我们的参数服务器通过确认这些关键值项的基本含义来改进此基本方法：机器学习算法通常将模型视为线性代数对象。例如，通过风险最小化，w 被用作算法 1 中的目标函数和优化的向量。通过将这些对象视为稀疏线性代数对象，参数服务器可以提供与(key,value)抽象相同的功能，但允许重要的优化操作，如向量相加 w + u，乘法 Xw，找到 2-范数$\| w \|_2$，以及其他更复杂的操作。

为了支持这些优化，我们假定 key 值是有序的。这让我们将参数视为(key,value)键值对，同时赋予它们矢量和矩阵语义，其中不存在键与零相关联。这有助于机器学习中的线性代数。它减少了实现优化算法的编程工作量。除了方便之外，这种接口设计通过利用 CPU 高效的多线程自调整线性代数库（例如 BLAS，LAPACK 和 ATLAS）来实现高效的代码。

3.2 范围推送和拉取

使用推拉操作在节点之间发送数据。在算法 1 中，每个 worker 将其整个本地梯度量推送到服务器，然后将更新的权重取回。算法 3 中描述的更高级的算法使用相同的模式，除了每次只传送范围内的 key 值。

参数服务器通过支持基于范围的推送和拉取来优化这些更新，以方便程序员以及计算和网络带宽效率。如果 R 是一个 key 值范围，则 w.push(R，dest)将 key 范围 R 中的所有 w 的现有条目发送到目标，该目标可以是特定节点，也可以是服务器组等节点组。类似地，w.pull（R，dest）从目标中读取 key 值范围 R 中的 w 的所有现有条目。如果我们将 R 设置为整个关键字范围，那么将传送整个向量 w。如果我们将 R 设置为包含单个 key，那么只会发送一个单独的条目。

该接口可以扩展为与任何本地数据结构进行通信，这些数据共享与 w 相同的 key。例如，在算法 1 中，worker 将其临时本地梯度 g 推送到参数服务器进行聚合。一种选择是使 g 全局共享。但是，请注意，g 共享 worker 工作集 w 的 key 值。因此，程序员可以使用 w.push(R，g，dest)作为本地梯度来节省内存，并且还可以享受以下各节讨论的优化。

Algorithm 3 Delayed Block Proximal Gradient [31]

Scheduler:

1: Partition features into b ranges $\mathcal{R}_1, \ldots, \mathcal{R}_b$
2: **for** $t = 0$ **to** T **do**
3: Pick random range $\mathcal{R}_{i_t}$ and issue task to workers
4: **end for**

Worker r at iteration t

1: Wait until all iterations before $t - \tau$ are finished
2: Compute first-order gradient $g_r^{(t)}$ and diagonal second-order gradient $u_r^{(t)}$ on range $\mathcal{R}_{i_t}$
3: Push $g_r^{(t)}$ and $u_r^{(t)}$ to servers with the KKT filter
4: Pull $w_r^{(t+1)}$ from servers

Servers at iteration t

1: Aggregate gradients to obtain $g^{(t)}$ and $u^{(t)}$
2: Solve the proximal operator

$$w^{(t+1)} \leftarrow \operatorname*{argmin}_u \Omega(u) + \frac{1}{2\eta}\|w^{(t)} - \eta g^{(t)} + u\|_H^2,$$

where $H = \operatorname{diag}(h^{(t)})$ and $\|x\|_H^2 = x^T H x$

3.3 基于服务器的用户定义功能

除了汇集来自工作者的数据之外，服务器节点可以执行用户定义的功能。这是有益的，因为服务器节点通常具有关于共享参数的更完整或最新的信息。在算法 1 中，服务器节点评估正规化器 Ω 的次梯度以便更新 w。 同时，一个更复杂的近端算子被服务器解决以更新算法 3 中的模型。在草图（5.3 节）的情况下，几乎所有的操作都发生在服务器端。

3.4 异步任务与依赖

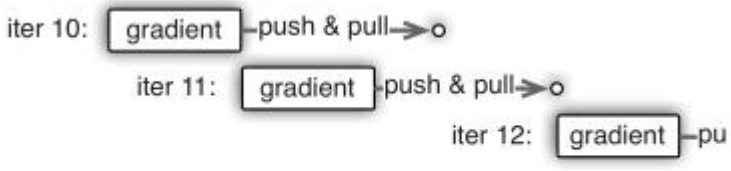

Figure 5: Iteration 12 depends on 11, while 10 and 11 are independent, thus allowing asynchronous processing.

任务由远程程序调用。它可能是 worker 向服务器发出的推送或拉取。它也可以是调度程序发布给任何节点的用户定义函数。任务可能包括任何数量的子任务。例如，算法 1 中的任务 WorkerIterate 包含一个推送和一个拉取。

任务是异步执行的：调用者可以在发出任务后立即进行进一步的计算。

调用者只有在收到被叫方的回复后才将其标记为已完成。回复可以是用户定义函数的函数返回，拉请求的(key,value)键值对或空的确认。被调用者只有在任务调用结束返回并且由该调用发出的所有子任务完成时才将任务标记为已完成。

默认情况下，被调用者并行执行任务，以获得最佳性能。希望序列化任务执行的调用者可以在任务之间放置执行完成后的依赖关系。Figure 5 描述了 WorkerIterate 的三个示例迭代。迭代 10 和 11 是独立的，但是 12 取决于 11.因此，被调用者在迭代 10 中计算局部梯度时立即开始第 11 次迭代。然而迭代 12 被推迟到 11 的结束完成。

任务依赖性有助于实现算法逻辑。例如，算法 1 的 ServerIterate 中的聚合逻辑仅在聚合所有 worker 的梯度后才更新权重 w。这可以通过使更新任务取决于所有 worker 的推送任务来实现。依赖关系的第二个重要用途是支持下面描述的灵活一致性模型。

3.5 灵活一致性

独立任务通过并行使用 CPU，磁盘和网络带宽来提高系统效率。但是，这可能会导致节点之间的数据不一致。在上图中，工作者 r 在 w(11)被拉回之前开始迭代 11，因此它在该迭代中使用旧的 w(10)，并因此获得与迭代 10 中相同的梯度，即 gr(11)=gr(10)。这种不一致可能会减慢算法 1 的收敛进度。然而，一些算法对这种不一致性可能不太敏感。例如，在算法 3 中每次只更新一段 w。因此，在不等待 10 的情况下开始迭代 11 只会导致一部分 w 不一致。

系统效率和算法收敛速度之间的最佳平衡通常取决于多种因素，包括算法对数据不一致性的敏感性，训练数据中的特征相关性以及硬件组件的容量差异。不是强迫用户采用一个可能会导致问题的特定依赖关系，而是参数服务器为算法设计者提供了定义一致性模型的灵活性。这与其他机器学习系统有很大的不同。

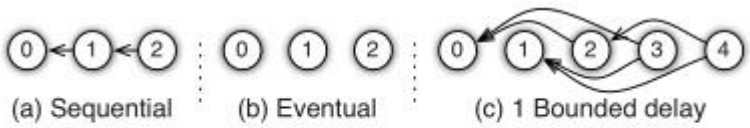

Figure 6: Directed acyclic graphs for different consistency models. The size of the DAG increases with the delay.

我们展示了三种可以通过任务依赖实现的不同模型。它们的相关有向无环图在 Figure 6 中给出。

顺序 在顺序一致性中，所有任务都是逐个执行的。 下一个任务只有在前一个任务完成时才能启动。它产生的结果与单线程实现相同，也称为批量同步处理。

最终性 最终的一致性是相反的：所有任务可以同时开始。 例如，描述这样一个系统。但是，如果基础算法在延迟方面是健壮的，那么这只是值得推荐的。

有界延迟 当最大延迟时间 τ 被设置时，一个新任务将被阻塞，直到 τ 时间前的所有先前任务完成。算法 3 使用这样的模型。这个模型比前两个模型提供了更灵活的控制： $\tau = 0$ 是顺序一致性模型，无限延迟 $\tau = \infty$ 变为最终的一致性模型。

请注意，依赖关系图可能是动态的。例如，调度器可以根据运行时进度来增加或减少最大延

迟，以平衡系统效率和底层优化算法的收敛。在这种情况下，调用者遍历 DAG。如果图形是静态的，则调用者可以将所有带 DAG 的任务发送给被调用者以降低同步成本。

3.6 用户定义的过滤器

作为基于调度程序流式控制的补充，参数服务器支持用户定义的过滤器，以有选择地同步各个(key,value)键值对，从而对任务内的数据一致性进行细粒度控制。优化算法本身通常需要分析出哪些参数对同步信息最有用。一个例子是重要的过滤器，它只推送自上次同步以来改变了超过阈值的条目。在 5.1 节中，我们讨论另一个名为 KKT 的过滤器，它利用了优化问题的最优性条件：**worker 只推动可能影响服务器权重的梯度。**

4 实现

服务器使用一致性散列存储参数（键，值对）（第 4.3 节）。为了容错，条目使用链复制进行复制（第 4.4 节）。与以前的（键，值）系统不同，参数服务器针对基于范围的通信和基于范围的矢量时钟（第 4.1 节）进行了压缩优化。

4.1 矢量时钟

Algorithm 2 Set vector clock to t for range $\mathcal{R}$ and node i

1: **for** $\mathcal{S} \in \{\mathcal{S}_i : \mathcal{S}_i \cap \mathcal{R} \neq \emptyset, i = 1, \ldots, n\}$ **do**
2: **if** $\mathcal{S} \subseteq \mathcal{R}$ **then** $\mathrm{vc}_i(\mathcal{S}) \leftarrow t$ **else**
3: $a \leftarrow \max(\mathcal{S}^b, \mathcal{R}^b)$ and $b \leftarrow \min(\mathcal{S}^e, \mathcal{R}^e)$
4: split range $\mathcal{S}$ into $[\mathcal{S}^b, a), [a, b), [b, \mathcal{S}^e)$
5: $\mathrm{vc}_i([a, b)) \leftarrow t$
6: **end if**
7: **end for**

考虑到潜在复杂的任务依赖关系图和快速恢复的需要，**每个(key, value)对都与一个向量时钟关联，该向量时钟记录每个单独节点在该(key, value)对上的时间**。矢量时钟很方便，例如，用于跟踪聚合状态或拒绝重复发送的数据。然而，向量时钟的简单实现需要 O(nm)空间来处理 n 个节点和 m 个参数。由于数千个节点和数十亿个参数，这在内存和带宽方面是不可行的。

幸运的是，许多参数具有相同的时间戳（由参数服务器基于范围的通信模式产生）：如果节点将参数推送到一个范围内，则与该节点关联的参数的时间戳可能相同。因此，它们可以压缩成单个范围的矢量时钟。更具体地说，假设 vci（k）是节点 i 上的值 k 的时间。给定 key 值范围 R，距离矢量时钟 vci（R）= t 表示任意值 k∈ R，vci（k）= t。

最初，每个节点 i 只有一个范围矢量时钟。它覆盖整个参数 key 空间，其范围为 0 作为其初始时间戳。每个范围集合可以分割范围并创建至多 3 个新的向量时钟（参见算法 2）。令 k 为算法传递的唯一范围的总数，则最多有 O（mk）个向量时钟，其中 m 是节点的数量.k 通常远小于参数的总数。这显着减少了范围矢量时钟所需的空间。

4.2 消息

节点可以将消息发送到单个节点或节点组。消息由范围 R 中的(key,value)键值对列表和相关的范围向量时钟组成：

$$[vc(R),(k_1,v_1),...,(k_p,v_p)]\, k_j \in R \text{ and } j \in \{1,...p\}$$

这是参数服务器的**基本通信格式**，不仅用于共享参数，还用于任务。对于后者，(key,value)键值对可以采用格式 (task ID, arguments or return results)。

消息可能携带范围 R 内所有可用 key 的子集。缺少的 key 将被分配相同的时间戳，而不会更改其值。消息可以按 key 范围分割。这发生在 worker 向整个服务器组发送消息或接收方节点的 key 分配发生更改时发生。通过这样做，我们划分(key,value)列表并分割距离矢量时钟，类似于算法 2。

由于机器学习问题通常需要高带宽，所以需要消息压缩。训练数据在迭代之间通常保持不变。worker 可能会再次发送相同的 key 列表。因此，接收节点需要缓存 key 列表。稍后，发送者只需发送列表的散列而不是列表本身。值又可以包含许多零条目。例如，在 5.1 节中评估的大部分参数在稀疏逻辑回归中保持不变。同样，用户定义的过滤器也可以将大部分值清零（参见 Figure 12）。因此我们只需要发送非零(key,value)键值对。我们使用快速的 Snappy 压缩库来压缩消息，有效地消除零。请注意，键缓存和值压缩可以联合使用。

4.3 一致性哈希

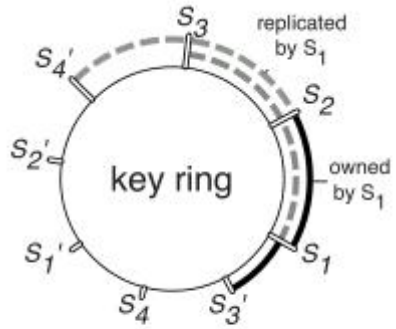

Figure 7: Server node layout.

参数服务器按照常规分布式哈希表所做的那样对 key 值进行分区：将 key 和服务器节点 ID 都插入哈希环中（Figure 7）。每个服务器节点以逆时针方向管理其插入点到下一个其他节点开始的键范围。这个节点被称为这个键的主范围。物理服务器通常通过多个“虚拟”服务器在环中表示，以改善负载平衡和复原。

我们通过使用直接映射的 DHT 设计来简化管理。服务器管理器处理环管理。所有服务器节点都在本地缓存 key 分区。通过这种方式，他们可以直接确定哪个服务器负责 key 范围，并通知任何更改。

4.4 复制和一致性

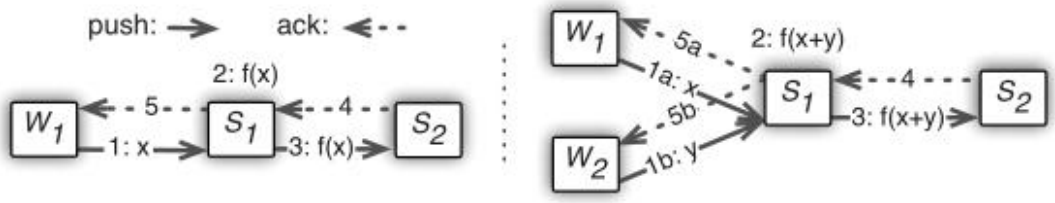

Figure 8: Replica generation. Left: single worker. Right: multiple workers updating values simultaneously.

每个服务器节点都存储相对于其拥有的 k 个逆时针方向的相邻 key 范围的副本。我们指的是将拷贝适当 key 范围的节点。上图显示了一个 k = 2 的示例，其中服务器 1 复制服务器 2 和服务器 3 拥有的范围内的 key 值。

worker 节点与推送拉取 key 值范围的 master 通信。master 上的任何修改都会将其时间戳复制到从设备。对数据的修改会同步推送到从设备。Figure 8 显示了 worker 1 将 x 推送到服务器 1 的情况，该服务器调用用户定义的函数 f 来修改共享数据。只有将数据修改 f(x)复制到从设备后，才能完成推送任务。

简单复制可能会将网络流量增加 k 倍。对于许多依靠高网络带宽的机器学习应用来说，这是不可取的。参数服务器框架允许对许多算法进行重要优化：聚合后复制。服务器节点通常会聚合来自 worker 节点的数据，例如汇总本地梯度。因此，服务器可能会推迟复制，直到聚合完成。在图的右侧，两名 worker 分别将 x 和 y 推送到服务器。**服务器首先通过 x + y 聚合推送的值**，然后应用修改 f(x + y)，最后执行复制。对于 n 个 worker，复制只使用 k / n 带宽。通常 k 是一个小常数，而 n 是几百到几千。虽然聚合增加了任务回复的延迟，但它可以通过宽松的一致性条件来隐藏。

4.5 服务器管理

要实现**容错和动态扩展**，我们必须支持添加和删除节点。为方便起见，我们参考下面的虚拟服务器，服务器加入时会发生以下步骤。

1. 服务器管理员为新节点分配一个 key 值范围以充当主节点。这可能会导致另一个 key 值范围分裂或从被终止的节点中删除。
2. 新节点从其他节点上将属于它的 key range 数据取过来，然后也将 slave 信息取过来
3. 服务器管理器广播节点更改信息。消息的接收节点可能会根据它们不再拥有的 key 值范围收缩自己的数据，并将未完成的任务重新提交给新节点。

从某节点 S 获取范围 R 中的数据分两个阶段进行，类似于 Ouroboros 协议。首先 S 预先复制范围内的所有(key,value)键值对以及相关的矢量时钟。这可能会导致范围向量时钟分裂，类似于算法 2。如果新节点在此阶段失败，则 S 保持不变。在第二阶段，S 不再接受影响 key 值范围 R 的消息，通过丢弃消息而不执行和回复。同时，S 在预复制阶段向新节点发送 R 中现阶段发生的所有更改。

在接收到节点改变消息时，节点 N 首先检查它是否也包含 key 范围 R。如果是，并且如果该 key 范围不再由 N 维持，则它删除所有关联的(key,value)键值对和向量时钟 R。接下来，N 扫描所有尚未收到回复的已传出的消息。如果 key 范围与 R 相交，则该消息将被拆分并重新发

送。

由于延迟，失败和丢失的确认，N 可能发送消息两次。由于使用矢量时钟，原始接收节点和新节点都能够拒绝此消息，并且不会影响正确性。

服务器节点的删除（自愿或失败）与加入类似。服务器管理器使用离开节点的关键字范围来执行新节点。服务器管理器通过心跳信号检测节点故障。与集群资源管理器（如 Yarn 或 Mesos）的集成留待将来进行。

4.6 工作节点管理

添加新的工作节点 W 与添加新的服务器节点相似，但更简单：

1. 任务调度程序为 W 分配一系列数据。
2. 该节点加载来自网络文件系统或现有 worker 的训练数据。训练数据通常是只读的，所以没有两阶段提取。接下来，W 从服务器获取共享参数。
3. 任务调度程序广播更改的信息，可能导致其他 worker 释放一些训练数据。

当一个 worker 节点删除时，任务调度器可以开始替换。 我们给算法设计者控制恢复的选项有两个原因：如果训练数据很大，恢复一个工作节点可能比恢复一个服务器节点代价更大。其次，在优化过程中丢失少量训练数据通常对模型影响不大。因此，算法设计者可能更喜欢继续而不替换失败的 worker，甚至可能希望终止最慢的 worker。