# Scaling Distributed Machine Learning with the Parameter Server

**Mu Li,** ***Carnegie Mellon University and Baidu;*** **David G. Andersen and Jun Woo Park,** ***Carnegie Mellon University;*** **Alexander J. Smola,** ***Carnegie Mellon University and Google, Inc.;*** **Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su,** ***Google, Inc.***

https://www.usenix.org/conference/osdi14/technical-sessions/presentation/li_mu

**This paper is included in the Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation.**

**October 6–8, 2014 • Broomfield, CO**

978-1-931971-16-4

**Open access to the Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation is sponsored by USENIX.**

usenix
THE ADVANCED
COMPUTING SYSTEMS
ASSOCIATION

# 使用参数服务器扩展分布式机器学习

穆立，卡内基梅隆大学和百度；大卫·G·安德森和金宇宇·帕克，卡内基梅隆大学；亚历山大·J·斯莫拉，卡内基梅隆大学和谷歌，Inc.；阿姆尔·阿赫梅德、万贾·约西福夫斯基、詹姆斯·朗、尤金·J·谢基塔和苏伯英，谷歌，Inc.

https://www.usenix.org/conference/osdi14/technical-sessions/presentation/li_mu

**本文被收录在《第11届USENIX操作系统设计与实现研讨会论文集》中**

**第11届USENIX操作系统设计与实现研讨会论文集**

**操作系统设计与实现。**

**10月6日至8日， 2014 • 布鲁姆菲尔德，科罗拉多州**

978-1-931971-16-4

**USENIX 赞助了第 11 届 USENIX 操作系统设计与实现研讨会论文的开放获取。**

# Scaling Distributed Machine Learning with the Parameter Server

Mu Li*‡, David G. Andersen*, Jun Woo Park*, Alexander J. Smola*†, Amr Ahmed†,
Vanja Josifovski†, James Long†, Eugene J. Shekita†, Bor-Yiing Su†

*Carnegie Mellon University ‡Baidu †Google

{muli, dga, junwoop}@cs.cmu.edu, alex@smola.org, {amra, vanjaj, jamlong, shekita, boryiingsu}@google.com

## Abstract

We propose a parameter server framework for distributed machine learning problems. Both data and workloads are distributed over worker nodes, while the server nodes maintain globally shared parameters, represented as dense or sparse vectors and matrices. The framework manages asynchronous data communication between nodes, and supports flexible consistency models, elastic scalability, and continuous fault tolerance.

To demonstrate the scalability of the proposed framework, we show experimental results on petabytes of real data with billions of examples and parameters on problems ranging from Sparse Logistic Regression to Latent Dirichlet Allocation and Distributed Sketching.

## 1 Introduction

Distributed optimization and inference is becoming a prerequisite for solving large scale machine learning problems. At scale, no single machine can solve these problems sufficiently rapidly, due to the growth of data and the resulting model complexity, often manifesting itself in an increased number of parameters. Implementing an efficient distributed algorithm, however, is not easy. Both intensive computational workloads and the volume of data communication demand careful system design.

Realistic quantities of training data can range between 1TB and 1PB. This allows one to create powerful and complex models with $10^9$ to $10^{12}$ parameters [9]. These models are often shared globally by all worker nodes, which must frequently accesses the shared parameters as they perform computation to refine it. Sharing imposes three challenges:

- Accessing the parameters requires an enormous amount of network bandwidth.
- Many machine learning algorithms are sequential. The resulting barriers hurt performance when the cost of synchronization and machine latency is high.
- At scale, fault tolerance is critical. Learning tasks are often performed in a cloud environment where machines can be unreliable and jobs can be preempted.

| ≈ #machine × 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.

To illustrate the last point, we collected all job logs for a three month period from one cluster at a large internet company. We show statistics of batch machine learning tasks serving a production environment in Table 1. Here, task failure is mostly due to being preempted or losing machines without necessary fault tolerance mechanisms.

Unlike in many research settings where jobs run exclusively on a cluster without contention, fault tolerance is a necessity in real world deployments.

### 1.1 Contributions

Since its introduction, the parameter server framework [43] has proliferated in academia and industry. This paper describes a third generation open source implementation of a parameter server that focuses on the systems aspects of distributed inference. It confers two advantages to developers: First, by factoring out commonly required components of machine learning systems, it enables application-specific code to remain concise. At the same time, as a shared platform to target for systems-level optimizations, it provides a robust, versatile, and high-performance implementation capable of handling a diverse array of algorithms from sparse logistic regression to topic models and distributed sketching. Our design de-



# 基于参数服务器扩展分布式机器学习

Mu Li*‡, David G. Andersen*, Jun Woo Park*, Alexander J. Smola*†, Amr Ahmed†,
Vanja Josifovski†, James Long†, Eugene J. Shekita†, Bor-Yiing Su†

*卡内基梅隆大学 ‡百度 †谷歌

{muli, dga, junwoop}@cs.cmu.edu, alex@smola.org, {amra, vanjaj, jamlong, shekita, boryiingsu}@谷歌.com

## 摘要

我们提出了一种用于分布式机器学习问题的参数服务器框架。数据和任务负载都分布在工作节点上，而服务器节点维护全局共享参数，这些参数表示为密集或稀疏的向量或矩阵。该框架管理节点之间的异步数据通信，并支持灵活一致性模型、弹性可扩展性和连续容错性。为了展示所提出框架的可扩展性，我们在真实数据上进行了实验，这些数据包含数十亿个示例和参数，问题范围从稀疏逻辑回归到潜在狄利克雷分配和分布式草稿。

## 1 引言

分布式优化和推理正成为解决大规模机器学习问题的一项先决条件。在规模上，由于数据增长和由此产生的模型复杂性（通常表现为参数数量的增加），没有一台单独的机器能够足够快速地解决这些问题。然而，实现高效的分布式算法并不容易。密集的计算工作负载和数据通信量都要求仔细的系统设计。

实际训练数据量可以在1TB到1PB之间。这使得人们可以用$10^9$ 到 $10^{12}$ 个参数 [9]创建强大而复杂的模型。这些模型通常由所有工作节点共享，它们在执行计算以优化参数时必须频繁访问共享参数。共享带来了三个挑战：

•访问参数需要巨大的网络带宽。

• 许多机器学习算法是顺序的。由此产生的障碍在需要时会影响性能。

| ≈ #机器 × 时间 | 作业数量 | 失败率 |
|---|---|---|
| 100小时 | 13,187 | 7.8% |
| 1,000小时 | 1,366 | 13.7% |
| 10,000小时 | 77 | 24.7% |

表1：数据中心三个月机器学习工作统计。

同步成本和机器延迟很高。

• 在规模化应用中，容错性至关重要。学习任务通常在云环境中执行，而机器可能不可靠且作业可能被抢占。

为了说明这一点，我们从一家大型互联网公司的一个集群收集了三个月的所有作业日志。我们在表1中展示了为生产环境服务的批量机器学习任务的统计数据。在这里，任务失败主要是由于被抢占或失去没有必要容错机制的机器。

与许多研究环境不同，在那些作业仅在一个无竞争的集群上运行的情况下，容错性在实际部署中是必要条件。

### 1.1 贡献

自其引入以来，参数服务器框架 [43] 已在学术界和工业界得到广泛应用。本文描述了一种第三代开源参数服务器实现，该实现专注于分布式推理的系统方面。它为开发者提供了两大优势：首先，通过将机器学习系统常用的组件进行解耦，使得针对特定应用的代码能够保持简洁。同时，作为一个用于系统级优化的共享平台，它提供了一种健壮、通用且高性能的实现，能够处理从稀疏逻辑回归到主题模型和分布式草图的各类算法。我们的设计

| | 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.

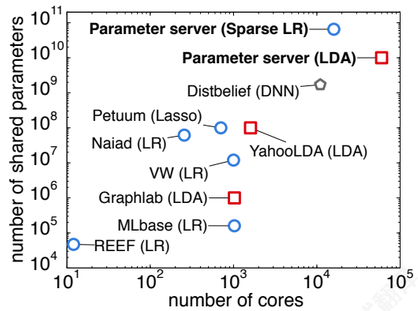


Figure 1: Comparison of the public largest machine learning experiments each system performed. Problems are color-coded as follows: Blue circles — sparse logistic regression; red squares — latent variable graphical models; grey pentagons — deep networks.

cisions were guided by the workloads found in real systems. Our parameter server provides five key features:

**Efficient communication:** The asynchronous communication model does not block computation (unless requested). It is optimized for machine learning tasks to reduce network traffic and overhead.

**Flexible consistency models:** Relaxed consistency further hides synchronization cost and latency. We allow the algorithm designer to balance algorithmic convergence rate and system efficiency. The best trade-off depends on data, algorithm, and hardware.

**Elastic Scalability:** New nodes can be added *without* restarting the running framework.

**Fault Tolerance and Durability:** Recovery from and repair of non-catastrophic machine failures within 1s, without interrupting computation. Vector clocks ensure well-defined behavior after network partition and failure.

**Ease of Use:** The globally shared parameters are represented as (potentially sparse) vectors and matrices to facilitate development of machine learning applications. The linear algebra data types come with high-performance multi-threaded libraries.

The novelty of the proposed system lies in the synergy achieved by picking the right systems techniques, adapting them to the machine learning algorithms, and modifying the machine learning algorithms to be more systems-friendly. In particular, we can relax a number of otherwise hard systems constraints since the associated machine learning algorithms are quite tolerant to perturbations. The consequence is the first general purpose ML system capable of scaling to industrial scale sizes.

## 1.2 Engineering Challenges

When solving distributed data analysis problems, the issue of reading and updating parameters shared between different worker nodes is ubiquitous. The parameter server framework provides an efficient mechanism for aggregating and synchronizing model parameters and statistics between workers. Each parameter server node maintains only a part of the parameters, and each worker node typically requires only a subset of these parameters when operating. Two key challenges arise in constructing a high performance parameter server system:

**Communication.** While the parameters could be updated as key-value pairs in a conventional datastore, using this abstraction naively is inefficient: values are typically small (floats or integers), and the overhead of sending each update as a key value operation is high.

Our insight to improve this situation comes from the observation that many learning algorithms represent parameters as structured mathematical objects, such as vectors, matrices, or tensors. At each logical time (or an iteration), typically a part of the object is updated. That is, workers usually send a *segment* of a vector, or an entire *row* of the matrix. This provides an opportunity to automatically batch both the communication of updates and their processing on the parameter server, and allows the consistency tracking to be implemented efficiently.

**Fault tolerance**, as noted earlier, is critical at scale, and for efficient operation, it must not require a full restart of a long-running computation. Live replication of parameters between servers supports hot failover. Failover and self-repair in turn support dynamic scaling by treating machine removal or addition as failure or repair respectively.

Figure 1 provides an overview of the scale of the largest supervised and unsupervised machine learning experiments performed on a number of systems. When possible, we confirmed the scaling limits with the authors of each of these systems (data current as of 4/2014). As is evident, we are able to cover orders of magnitude more data on orders of magnitude more processors than any



| | 共享数据 | 一致性 | 容错性 |
|---|---|---|---|
| Graphlab [34] | 图 | 最终一致性 | 检查点 |
| Petuum [12] | 哈希表 | 延迟界限 | none |
| REEF [10] | 数组 | BSP | 检查点 |
| Naiad [37] | (键,值) | 多个 | 检查点 |
| Mlbase [29] | 表格 | BSP | RDD |
| 参数（稀疏）服务器 | 向量/矩阵 | 各种 | 连续的 |

表 2：分布式数据分析系统的属性。

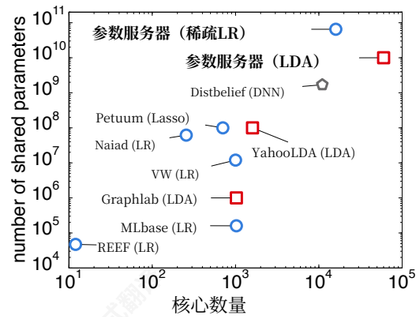


图1：比较每个系统执行过的公共最大机器学习实验。问题按以下颜色编码：蓝色圆圈 — 稀疏逻辑回归；红色方块 — 隐变量图模型；灰色五边形 — 深度网络。

决策是基于真实系统中发现的工作负载进行的。我们的参数服务器提供五个关键特征：

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

灵活一致性模型：宽松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。最佳权衡取决于数据、算法和硬件。

弹性可扩展性：新节点可以添加而无需重启运行中的框架。

容错性和持久性：在 1 秒内从非灾难性机器故障中恢复和修复，而不会中断计算。向量时钟确保在网络分区和故障后行为良好定义。

易用性：全局共享参数被表示为（潜在的稀疏）向量和矩阵，以方便开发机器学习应用。线性代数数据类型配备了高性能的多线程库。

所提出系统的创新之处在于通过选择合适的系统技术、将它们适配到机器学习算法，并修改机器学习算法使其更易于系统交互而实现的协同效应。特别是，我们可以放宽许多原本困难的系统约束，因为相关的机器学习算法对扰动相当容忍。其结果是首个能够扩展到工业规模的一般性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之间共享参数的问题普遍存在。参数服务器框架为聚合和同步工作节点之间的模型参数和统计数据提供了高效机制。每个参数服务器节点仅包含参数的一部分，而每个工作节点在运行时通常只需要这些参数的子集。

构建高性能参数服务器系统时会出现两个关键挑战：

通信。虽然参数可以作为键值对在传统数据存储中更新，但直接使用这种抽象效率低下：值通常很小（浮点数或整数），而将每个更新作为键值操作发送的开销很高。

我们改进这种情况的见解来自于观察到许多学习算法将参数表示为结构化的数学对象，例如向量、矩阵或张量。在每个逻辑时间（或迭代）中，对象的一部分通常会被更新。也就是说，工作者通常发送向量的一个片段，或矩阵的一整行。这为自动批量更新通信及其在参数服务器上的处理提供了机会，并允许高效地实现一致性跟踪。

如前所述，容错性在规模化应用中至关重要，而为了高效运行，它不能要求对长时间运行的计算进行完整重启。服务器之间参数的实时复制支持热故障切换。故障切换和自我修复反过来又支持动态扩展，将机器移除或添加视为故障或修复操作。

图1展示了在多个系统上执行的最大规模的监督学习和无监督机器学习实验的概述。在可能的情况下，我们与这些系统的作者确认了扩展极限（数据截至2014年4月）。很明显，我们能够在比任何其他已发表系统多几个数量级的处理器上处理多几个数量级的数据。

other published system. Furthermore, Table 2 provides an overview of the main characteristics of several machine learning systems. Our parameter server offers the greatest degree of flexibility in terms of consistency. It is the only system offering continuous fault tolerance. Its native data types make it particularly friendly for data analysis.

### 1.3 Related Work

Related systems have been implemented at Amazon, Baidu, Facebook, Google [13], Microsoft, and Yahoo [1]. Open source codes also exist, such as YahooLDA [1] and Petuum [24]. Furthermore, Graphlab [34] supports parameter synchronization on a best effort model.

The first generation of such parameter servers, as introduced by [43], lacked flexibility and performance — it repurposed `memcached` distributed (key,value) store as synchronization mechanism. YahooLDA improved this design by implementing a dedicated server with user-definable update primitives (set, get, update) and a more principled load distribution algorithm [1]. This second generation of *application specific* parameter servers can also be found in Distbelief [13] and the synchronization mechanism of [33]. A first step towards a general platform was undertaken by Petuum [24]. It improves YahooLDA with a bounded delay model while placing further constraints on the worker threading model. We describe a third generation system overcoming these limitations.

Finally, it is useful to compare the parameter server to more general-purpose distributed systems for machine learning. Several of them mandate synchronous, iterative communication. They scale well to tens of nodes, but at large scale, this synchrony creates challenges as the chance of a node operating slowly increases. Mahout [4], based on Hadoop [18] and MLI [44], based on Spark [50], both adopt the iterative MapReduce [14] framework. A key insight of Spark and MLI is preserving state between iterations, which is a core goal of the parameter server.

Distributed GraphLab [34] instead asynchronously schedules communication using a graph abstraction. At present, GraphLab lacks the elastic scalability of the map/reduce-based frameworks, and it relies on coarse-grained snapshots for recovery, both of which impede scalability. Its applicability for certain algorithms is limited by its lack of global variable synchronization as an efficient first-class primitive. In a sense, a core goal of the parameter server framework is to capture the benefits of GraphLab's asynchrony without its structural limitations.

Piccolo [39] uses a strategy related to the parameter server to share and aggregate state between machines. In it, workres pre-aggregate state locally and transmit the updates to a server keeping the aggregate state. It thus implements largely a subset of the functionality of our system, lacking the mechane learning specailized optimizations: message compression, replication, and variable consistency models expressed via dependency graphs.

## 2 Machine Learning

Machine learning systems are widely used in Web search, spam detection, recommendation systems, computational advertising, and document analysis. These systems automatically learn models from examples, termed *training data*, and typically consist of three components: *feature extraction*, the *objective function*, and *learning*.

Feature extraction processes the raw training data, such as documents, images and user query logs, to obtain *feature vectors*, where each feature captures an attribute of the training data. Preprocessing can be executed efficiently by existing frameworks such as MapReduce, and is therefore outside the scope of this paper.

### 2.1 Goals

The goal of many machine learning algorithms can be expressed via an "objective function." This function captures the properties of the learned model, such as low error in the case of classifying e-mails into ham and spam, how well the data is explained in the context of estimating topics in documents, or a concise summary of counts in the context of sketching data.

The learning algorithm typically minimizes this objective function to obtain the model. In general, there is no closed-form solution; instead, learning starts from an initial model. It iteratively refines this model by processing the training data, possibly multiple times, to approach the solution. It stops when a (near) optimal solution is found or the model is considered to be converged.

The training data may be extremely large. For instance, a large internet company using one year of an ad impression log [27] to train an *ad click predictor* would have trillions of training examples. Each training example is typically represented as a possibly very high-dimensional "feature vector" [9]. Therefore, the training data may consist of trillions of trillion-length feature vectors. Iteratively processing such large scale data requires enormous computing and bandwidth resources. Moreover, billions of new ad impressions may arrive daily. Adding this data into the system often improves both prediction accuracy and coverage. But it also requires the learning algorithm to run daily [35], possibly in real time. Efficient execution of these algorithms is the main focus of this paper.



此外，表2概述了几个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了最大的灵活性。它是唯一提供连续容错性的系统。其原生数据类型使其特别适合数据分析。

### 1.3 相关工作

相关系统已在亚马逊、百度、Facebook、谷歌 [13], 微软和雅虎 [1]实现。开源代码也存在，例如YahooLDA [1]和Petuum [24]。此外，Graphlab [34]支持在尽力而为的模型上进行参数同步。

第一代此类参数服务器，正如 [43], 所介绍的那样，缺乏灵活性和性能——它将memcached分布式（键值）存储改用为同步机制。YahooLDA通过实现具有用户定义更新原语（设置、获取、更新）和更原则性的负载分配算法 [1]改进了此设计。这种特定于应用的第二代参数服务器也可以在Distbelief [13] 和 [33]的同步机制中找到。Petuum [24]为通用平台迈出了第一步。它在为工作者线程模型施加进一步约束的同时，通过有界延迟模型改进了YahooLDA。我们描述了一个克服这些限制的第三代系统。

最后，将参数服务器与更通用的机器学习分布式系统进行比较是很有用的。其中一些系统要求同步、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，这种同步性会带来挑战，因为节点运行缓慢的可能性增加了。基于 Hadoop [18] 的 Mahout [4], 和基于 Spark [50], 的 MLI [44], 都采用了迭代 MapReduce [14] 框架。Spark 和 MLI 的一个关键洞察是在迭代之间保留状态，这是参数服务器的核心目标。

分布式GraphLab [34]采用图抽象异步调度通信。目前，GraphLab缺乏基于map/reduce的框架的弹性可扩展性，并且它依赖粗粒度快照进行恢复，这两者都阻碍了可扩展性。由于缺乏作为高效第一类原语的全局变量同步，其在某些算法上的适用性受到限制。从某种意义上说，参数服务器框架的一个核心目标是在不引入GraphLab结构限制的前提下，捕获其异步性的优势。

Piccolo [39]采用与参数服务器相关的策略，在机器之间共享和聚合状态。在其中，工作节点在本地预先聚合状态，并将聚合状态传输到保存聚合状态的服务器。

它实现了我们系统功能的大致子集，缺乏机器学习专门优化的机制：消息压缩、复制和通过依赖图表达的变量一致性模型。

## 2 机器学习

机器学习系统被广泛应用于网络搜索、垃圾邮件检测、推荐系统、计算广告和文档分析等领域。这些系统自动从示例（称为训练数据）中学习模型，通常由三个组件构成：特征提取、目标函数和学习过程。

特征提取过程处理原始训练数据，例如文档、图像和用户查询日志，以获得特征向量，其中每个特征捕获训练数据的一个属性。预处理可以通过现有的框架（如MapReduce）高效执行，因此不在此论文的范围内。

### 2.1 目标

许多机器学习算法的目标可以通过“目标函数”来表达。该函数捕获学习模型的属性，例如在将电子邮件分类为正常邮件和垃圾邮件时低误差，在文档主题估计的上下文中数据解释得如何，或在绘制数据时对计数的简洁总结。

学习算法通常通过最小化这个目标函数来获取模型。一般来说，没有封闭形式的解；相反，学习从初始模型开始。它通过处理训练数据（可能多次）来迭代地优化这个模型，以逼近解。当找到一个（近）最优解或模型被认为收敛时，它就会停止。

训练数据可能非常庞大。例如，一个大型互联网公司使用一年的广告展示日志 [27] 来训练一个广告点击预测器，将拥有数万亿个训练样本。每个训练样本通常表示为一个可能非常高维的“特征向量”[9]。因此，训练数据可能由数万亿个万亿长度的特征向量组成。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，每天可能有数十亿新的广告展示。将这些数据添加到系统中通常会提高预测准确性和覆盖范围。但它也需要学习算法每天 [35], 可能实时运行。这些算法的高效执行是本文的主要关注点。

To motivate the design decisions in our system, next we briefly outline the two widely used machine learning technologies that we will use to demonstrate the efficacy of our parameter server. More detailed overviews can be found in [36, 28, 42, 22, 6].

## 2.2 Risk Minimization

The most intuitive variant of machine learning problems is that of risk minimization. The "risk" is, roughly, a measure of prediction error. For example, if we were to predict tomorrow's stock price, the risk might be the deviation between the prediction and the actual value of the stock.

The training data consists of $n$ examples. $x_i$ is the $i$th such example, and is often a vector of length $d$. As noted earlier, both $n$ and $d$ may be on the order of billions to trillions of examples and dimensions, respectively. In many cases, each training example $x_i$ is associated with a label $y_i$. In ad click prediction, for example, $y_i$ might be 1 for "clicked" or -1 for "not clicked".

Risk minimization learns a model that can predict the value $y$ of a future example $x$. The model consists of parameters $w$. In the simplest example, the model parameters might be the "clickiness" of each feature in an ad impression. To predict whether a new impression would be clicked, the system might simply sum its "clickiness" based upon the features present in the impression, namely $x^\top w := \sum_{j=1}^{d} x_j w_j$, and then decide based on the sign.

In any learning algorithm, there is an important relationship between the amount of training data and the model size. A more detailed model typically improves accuracy, but only up to a point: If there is too little training data, a highly-detailed model will *overfit* and become merely a system that uniquely memorizes every item in the training set. On the other hand, a too-small model will fail to capture interesting and relevant attributes of the data that are important to making a correct decision.

Regularized risk minimization [48, 19] is a method to find a model that balances model complexity and training error. It does so by minimizing the sum of two terms: a *loss* $\ell(x, y, w)$ representing the prediction error on the training data and a *regularizer* $\Omega[w]$ penalizing the model complexity. A good model is one with low error and low complexity. Consequently we strive to minimize

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

The specific loss and regularizer functions used are important to the prediction performance of the machine learning algorithm, but relatively unimportant for the purpose of this paper: the algorithms we present can be used with all of the most popular loss functions and regularizers.

In Section 5.1 we use a high-performance distributed learning algorithm to evaluate the parameter server. For the sake of simplicity we describe a much simpler model

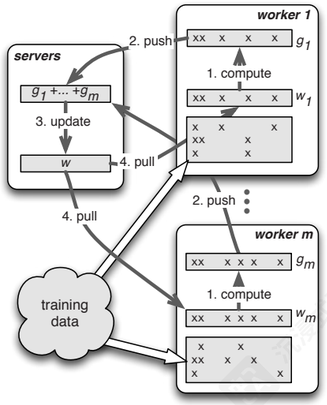


Figure 2: Steps required in performing distributed subgradient descent, as described e.g. in [46]. Each worker only caches the working set of $w$ rather than all parameters.

**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**



为了说明我们系统设计决策的动机，接下来我们将简要概述我们将用于演示参数服务器有效性的两种广泛使用的机器学习技术。更详细的概述可以在 [36, 28, 42, 22, 6] 中找到。

## 2.2 风险最小化

机器学习问题中最直观的变体是风险最小化。"风险"大致上是预测误差的度量。例如，如果我们预测明天的股价，风险可能是预测值与股票实际值之间的偏差。

训练数据由 $n$ 个示例组成。 $x_i$ 是第 $i$个这样的示例，通常是一个长度为 $d$的向量。如前所述， $n$ 和 $d$ 的数量级可能分别为数十亿到数万亿个示例和维度。在许多情况下，每个训练示例$x_i$ 都与一个标签$y_i$相关联。例如，在广告点击预测中， $y_i$ 可能是"点击"的1或"未点击"的-1。

风险最小化学习一个可以预测未来示例 $x$的值 $y$ 的模型。该模型由参数 $w$组成。在最简单的例子中，模型参数可能是广告展示中每个特征的"点击率"。为了预测新的展示是否会被点击，系统可能会根据展示中存在的特征，即$x^\top w := \sum_{j=1}^{d} x_j w_j$，简单地对其"点击率"求和，然后根据符号做出决定。

在任意学习算法中，训练数据量与模型大小之间存在重要关系。更详细的模型通常会提高准确率，但仅限于某个程度：如果训练数据太少，高度详细的模型会过拟合，仅仅变成一个能独特记忆训练集中每个项目的系统。另一方面，模型太小则无法捕捉数据中那些对做出正确决策很重要的有趣且相关的属性。

正则化风险最小化 [48, 19]是一种平衡模型复杂度和训练误差的方法。它通过最小化两项之和来实现：一项是代表训练数据预测误差的损失 $\ell(x, y, w)$，另一项是惩罚模型复杂度的正则化器 $\Omega[w]$ 。一个好的模型是误差低且复杂度低的模型。因此我们致力于最小化

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

具体的损失函数和正则化器对机器学习算法的预测性能很重要，但对本论文的目的而言相对不重要：

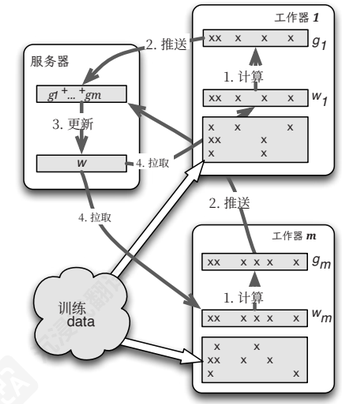


图2：执行分布式子图梯度下降所需的步骤，例如在 [46]中描述。每个工作节点仅缓存 $w$ 的工作集，而不是所有参数。

**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**

我们提出的算法可以与所有最流行的损失函数和正则化器一起使用。

在 5.1 节中，我们使用一种高性能的分布式学习算法来评估参数服务器。为了简化起见，我们描述了一个远为简单的模型

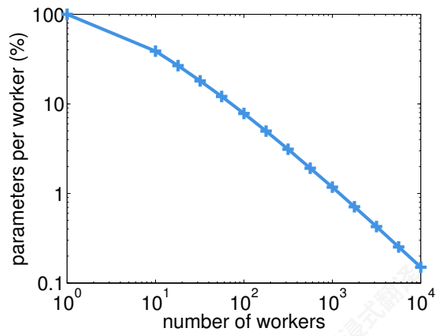


Figure 3: Each worker's set of parameters shrinks as more workers are used, requiring less memory per machine.

[46] called *distributed subgradient descent*.[1]

As shown in Figure 2 and Algorithm 1, the training data is partitioned among all of the workers, which jointly learn the parameter vector $w$. The algorithm operates iteratively. In each iteration, every worker independently uses its own training data to determine what changes should be made to $w$ in order to get closer to an optimal value. Because each worker's updates reflect only its own training data, the system needs a mechanism to allow these updates to mix. It does so by expressing the updates as a subgradient—a direction in which the parameter vector $w$ should be shifted—and aggregates all subgradients before applying them to $w$. These gradients are typically scaled down, with considerable attention paid in algorithm design to the right *learning rate* $\eta$ that should be applied in order to ensure that the algorithm converges quickly.

The most expensive step in Algorithm 1 is computing the subgradient to update $w$. This task is divided among all of the workers, each of which execute WORKERITERATE. As part of this, workers compute $w^\top x_{i_k}$, which could be infeasible for very high-dimensional $w$. Fortunately, a worker needs to know a coordinate of $w$ if and only if some of its training data references that entry.

For instance, in ad-click prediction one of the key features are the words in the ad. If only very few advertisements contain the phrase *OSDI 2014*, then most workers will not generate any updates to the corresponding entry in $w$, and hence do not require this entry. While the *total* size of $w$ may exceed the capacity of a single machine, the working set of entries needed by a particular worker can be trivially cached locally. To illustrate this, we randomly assigned data to workers and then counted the average working set size per worker on the dataset that is used in Section 5.1. Figure 3 shows that for 100 workers, each worker only needs 7.8% of the total parameters. With 10,000 workers this reduces to 0.15%.

## 2.3 Generative Models

In a second major class of machine learning algorithms, the label to be applied to training examples is unknown. Such settings call for *unsupervised* algorithms (for labeled training data one can use *supervised* or *semi-supervised* algorithms). They attempt to capture the underlying structure of the data. For example, a common problem in this area is *topic modeling*: Given a collection of documents, infer the topics contained in each document.

When run on, e.g., the SOSP'13 proceedings, an algorithm might generate topics such as "distributed systems", "machine learning", and "performance." The algorithms infer these topics from the content of the documents themselves, not an external topic list. In practical settings such as content personalization for recommendation systems [2], the scale of these problems is huge: hundreds of millions of users and billions of documents, making it critical to parallelize the algorithms across large clusters.

Because of their scale and data volumes, these algorithms only became commercially applicable following the introduction of the first-generation parameter servers [43]. A key challenge in topic models is that the parameters describing the current estimate of how documents are supposed to be generated must be shared.

A popular topic modeling approach is Latent Dirichlet Allocation (LDA) [7]. While the statistical model is quite different, the resulting algorithm for learning it is very similar to Algorithm 1.[2] The key difference, however, is that the update step is not a gradient computation, but an estimate of how well the document can be explained by the current model. This computation requires access to auxiliary metadata for each document that is updated each time a document is accessed. Because of the number of documents, metadata is typically read from and written back to disk whenever the document is processed.

This auxiliary data is the set of topics assigned to each word of a document, and the parameter $w$ being learned consists of the relative frequency of occurrence of a word.

As before, each worker needs to store only the parameters for the words occurring in the documents it processes. Hence, distributing documents across workers has

[1]The unfamiliar reader could read this as *gradient descent*; the subgradient aspect is simply a generalization to loss functions and regularizers that need not be continuously differentiable, such as $|w|$ at $w = 0$.

[2]The specific algorithm we use in the evaluation is a parallelized variant of a stochastic variational sampler [25] with an update strategy similar to that used in YahooLDA [1].



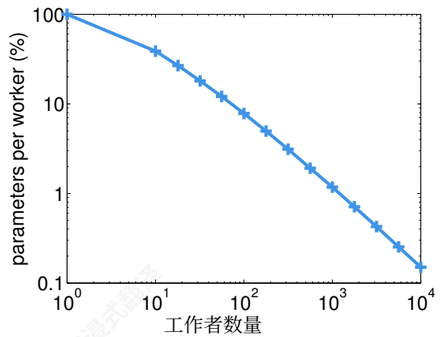


图3：随着使用更多工作者，每个工作者的参数集会缩小，从而减少每台机器的内存需求。

[46] 称为分布式子梯度下降算法。[1]

如图2和算法1所示，训练数据被分配给所有工作者，它们联合学习参数向量 $w$。该算法采用迭代方式运行。在每次迭代中，每个工作者独立使用自己的训练数据来确定如何修改 $w$ 以更接近最优值。由于每个工作者的更新仅反映其自身的训练数据，系统需要一个机制来允许这些更新混合。它通过将更新表示为参数向量 $w$应移动的方向——即子梯度——并将所有子梯度聚合后再应用于 $w$来实现这一点。这些梯度通常会进行缩放，算法设计中会仔细考虑正确的学习率 $\eta$，以确保算法能够快速收敛。

算法1中最昂贵的步骤是计算子梯度以更新 $w$。这项任务由所有工作者分担，每个工作者都执行WORKERIT-ERATE。作为其中一部分，工作者计算 $w^\top x_{i_k}$，对于非常高维度的 $w$来说这可能不可行。幸运的是，如果某个工作者的训练数据引用了某个条目，那么它只需要知道 $w$ 的一个坐标。

例如，在广告点击预测中，广告中的关键词是其中一个关键特征。如果只有极少数广告包含短语"OSDI 2014"，那么大多数工作者都不会生成任何更新到 $w$ 中对应的条目，因此不需要这个条目。虽然 $w$ 的总大小可能超过单台机器的容量，但特定工作者所需的条目工作集可以轻松本地缓存。为了说明这一点，我们为工作者分配了数据，然后在 5.1 节使用的数据集上计算了每个工作者的平均工作集大小。图 3 显示，对于 100 个工作者，每个工作者只需要总参数的 7.8%。如果有 10,000 个工作者，这个比例会减少到 0.15%。

[1]不熟悉的读者可能会将其读作梯度下降；子梯度方面只是对不需要连续可微分的损失函数和正则化器的一般化，例如在 $w = 0$处的 $|w|$。

我们简单地分配数据给工作者，然后在 5.1 节使用的数据集上计算每个工作者的平均工作集大小。图 3 显示，对于 100 个工作者，每个工作者只需要总参数的 7.8%。如果有 10,000 个工作者，这个比例会减少到 0.15%。

## 2.3 生成模型

在机器学习算法的第二大类中，应用于训练样本的标签是未知的。这种情况下需要无监督算法（对于标记的训练数据可以使用监督或半监督算法）。它们试图捕获数据的底层结构。例如，该领域的一个常见问题是主题建模：给定一组文档，推断每篇文档中包含的主题。

当在例如 SOSP' 13 会议录上运行时，一个算法可能会生成诸如"分布式系统"、"机器学习"和"性能"等主题。这些算法是从文档本身的 内容中推断出这些主题的，而不是从外部主题列表中推断。在实际场景中，例如用于推荐系统的个性化内容[2]，，这些问题的规模是巨大的：数以百万计的用户和数以亿计的文档，这使得跨大型集群并行化这些算法至关重要。

由于它们的规模和数据量，这些算法在第一代参数服务器 [43] 引入后才变得商业上可行。主题模型中的一个关键挑战是，必须共享描述当前估计文档应该如何生成的参数。

一个流行的主题建模方法是潜在狄利克雷分配（LDA) [7]。虽然统计模型差异很大，但学习它的算法与算法 1.[2] 非常相似。然而，主要区别在于更新步骤不是梯度计算，而是估计当前模型如何解释文档。这种计算需要访问每个文档的辅助元数据，每次访问文档时都会更新这些元数据。由于文档数量众多，在处理每个文档时，元数据通常需要从磁盘读取并写回磁盘。

这些辅助数据是分配给文档每个单词的主题集合，而学习的参数 $w$ 包括一个单词出现的相对频率。

如前所述，每个工作者只需存储其处理的文档中出现的参数。因此，将文档分发给工作者与

[2]我们在评估中使用的具体算法是一种并行化的随机变分采样器 [25]，其更新策略与YahooLDA [1]中使用的类似。

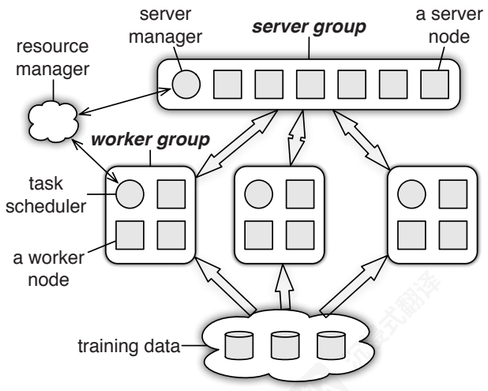


Figure 4: Architecture of a parameter server communicating with several groups of workers.

the same effect as in the previous section: we can process much bigger models than a single worker may hold.

## 3 Architecture

An instance of the parameter server can run more than one algorithm simultaneously. Parameter server nodes are grouped into a server group and several worker groups as shown in Figure 4. A server node in the server group maintains a partition of the globally shared parameters. Server nodes communicate with each other to replicate and/or to migrate parameters for reliability and scaling. A server manager node maintains a consistent view of the metadata of the servers, such as node liveness and the assignment of parameter partitions.

Each worker group runs an application. A worker typically stores locally a portion of the training data to compute local statistics such as gradients. Workers communicate only with the server nodes (not among themselves), updating and retrieving the shared parameters. There is a scheduler node for each worker group. It assigns tasks to workers and monitors their progress. If workers are added or removed, it reschedules unfinished tasks.

The parameter server supports independent parameter namespaces. This allows a worker group to isolate its set of shared parameters from others. Several worker groups may also share the same namespace: we may use more than one worker group to solve the same deep learning application [13] to increase parallelization. Another example is that of a model being actively queried by some nodes, such as online services consuming this model. Simultaneously the model is updated by a different group of worker nodes as new training data arrives.

The parameter server is designed to simplify developing distributed machine learning applications such as those discussed in Section 2. The shared parameters are presented as (key,value) vectors to facilitate linear algebra operations (Sec. 3.1). They are distributed across a group of server nodes (Sec. 4.3). Any node can both push out its local parameters and pull parameters from remote nodes (Sec. 3.2). By default, workloads, or tasks, are executed by worker nodes; however, they can also be assigned to server nodes via user defined functions (Sec. 3.3). Tasks are asynchronous and run in parallel (Sec. 3.4). The parameter server provides the algorithm designer with flexibility in choosing a consistency model via the task dependency graph (Sec. 3.5) and predicates to communicate a subset of parameters (Sec. 3.6).

### 3.1 (Key,Value) Vectors

The model shared among nodes can be represented as a set of (key, value) pairs. For example, in a loss minimization problem, the pair is a feature ID and its weight. For LDA, the pair is a combination of the word ID and topic ID, and a count. Each entry of the model can be read and written locally or remotely by its key. This (key,value) abstraction is widely adopted by existing approaches [37, 29, 12].

Our parameter server improves upon this basic approach by acknowledging the underlying meaning of these key value items: machine learning algorithms typically treat the model as a linear algebra object. For instance, $w$ is used as a vector for both the objective function (1) and the optimization in Algorithm 1 by risk minimization. By treating these objects as sparse linear algebra objects, the parameter server can provide the same functionality as the (key,value) abstraction, but admits important optimized operations such as vector addition $w + u$, multiplication $Xw$, finding the 2-norm $\|w\|_2$, and other more sophisticated operations [16].

To support these optimizations, we assume that the keys are ordered. This lets us treat the parameters as (key,value) pairs while endowing them with vector and matrix semantics, where non-existing keys are associated with zeros. This helps with linear algebra in machine learning. It reduces the programming effort to implement optimization algorithms. Beyond convenience, this interface design leads to efficient code by leveraging CPU-efficient multithreaded self-tuning linear algebra libraries such as BLAS [16], LAPACK [3], and ATLAS [49].



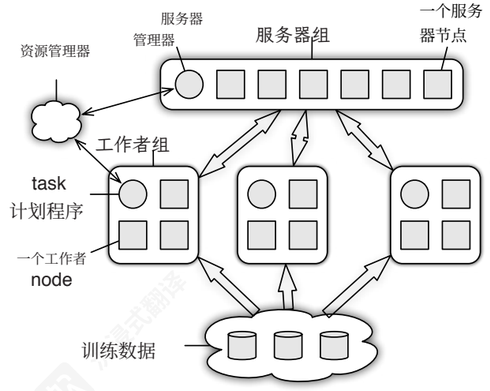


图4：参数服务器与多个工作者组通信的架构

节点，同时模型会由不同的一组工作节点更新，当新的训练数据到达时。

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。共享参数以（键，值）向量的形式呈现，以促进线性代数运算（第3.1节）。它们分布在服务器节点组中（第4.3节）。任何节点都可以将其本地参数推送到远程节点并从远程节点拉取参数（第3.2节）。默认情况下，工作负载或任务由工作节点执行；但是，它们也可以通过用户定义函数分配给服务器节点（第3.3节）。任务是异步的，并行运行（第3.4节）。参数服务器通过任务依赖关系图（第3.5节）为算法设计者提供选择一致性模型的灵活性，并通过谓词（第3.6节）来通信参数子集。

上一节中的效果相同：我们可以处理比单个工作者能容纳的更大的模型。

## 3 架构

参数服务器的一个实例可以同时运行多个算法。参数服务器节点被分组为服务器组和多个工作组，如图4所示。服务器组中的服务器节点维护全局共享参数的一个分区。服务器节点相互通信以复制和/或迁移参数，从而实现可靠性和可扩展性。服务器管理节点维护服务器元数据的一致视图，例如节点活性和参数分区的分配。

每个工作组运行一个应用程序。工作节点通常在本地存储训练数据的一部分，以计算局部统计量（如梯度）工作节点仅与服务器节点通信（彼此之间不通信），更新和检索共享参数。每个工作组有一个调度节点。它将任务分配给工作节点并监控其进度。如果添加或移除工作节点，它会重新调度未完成的任务。

参数服务器支持独立的参数命名空间。这使得一个工作节点组能够将其共享参数集与其他组隔离。多个工作节点组也可能共享相同的命名空间：我们可以使用多个工作节点组来解决相同的深度学习应用 [13] 以增加并行化。另一个例子是模型正被一些节点（例如在线服务）主动查询。同时，模型会由不同的一组工作节点更新，当新的训练数据到达时。

### 3.1 (键,值) 向量

模型在节点之间共享，可以表示为一组（键,值）对。例如，在损失最小化问题中，这对是一个特征ID及其权重。对于LDA，这对是词ID和主题ID的组合以及计数。模型的每个条目都可以通过其键在本地或远程读取和写入。这种（键,值）抽象被现有方法广泛采用 [37, 29, 12]。

我们的参数服务器通过承认这些键值项的潜在含义来改进这种基本方法：机器学习算法通常将模型视为线性代数对象。例如，$w$ 被用作风险最小化（1）的目标函数和算法1中的优化的向量。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（键,值）抽象相同的功能，但允许重要的优化操作，如向量加法 $w + u$、乘法 $Xw$、查找2-范数 $\|w\|_2$以及其他更复杂的操作 [16]。

为了支持这些优化，我们假设键是有序的。这使得我们可以将参数视为（键，值）对，同时赋予它们向量和矩阵语义，其中不存在的键与零相关联。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。除了便利性之外，这种接口设计通过利用CPU高效的、支持自调优的线性代数库（如BLAS [16], LAPACK [3],和ATLAS [49]）来生成高效的代码。

### 3.2 Range Push and Pull

Data is sent between nodes using `push` and `pull` operations. In Algorithm 1 each worker pushes its entire local gradient into the servers, and then pulls the updated weight back. The more advanced algorithm described in Algorithm 3 uses the same pattern, except that only a range of keys is communicated each time.

The parameter server optimizes these updates for programmer convenience as well as computational and network bandwidth efficiency by supporting *range-based* push and pull. If $\mathcal{R}$ is a key range, then `w.push(`$\mathcal{R}$`, dest)` sends all existing entries of $w$ in key range $\mathcal{R}$ to the destination, which can be either a particular node, or a node group such as the server group. Similarly, `w.pull(`$\mathcal{R}$`, dest)` reads all existing entries of $w$ in key range $\mathcal{R}$ from the destination. If we set $\mathcal{R}$ to be the whole key range, then the whole vector $w$ will be communicated. If we set $\mathcal{R}$ to include a single key, then only an individual entry will be sent.

This interface can be extended to communicate any local data structures that share the same keys as $w$. For example, in Algorithm 1, a worker pushes its temporary local gradient $g$ to the parameter server for aggregation. One option is to make $g$ globally shared. However, note that $g$ shares the keys of the worker's working set $w$. Hence the programmer can use `w.push(`$\mathcal{R}$`, g, dest)` for the local gradients to save memory and also enjoy the optimization discussed in the following sections.

### 3.3 User-Defined Functions on the Server

Beyond aggregating data from workers, server nodes can execute user-defined functions. It is beneficial because the server nodes often have more complete or up-to-date information about the shared parameters. In Algorithm 1, server nodes evaluate subgradients of the regularizer $\Omega$ in order to update $w$. At the same time a more complicated proximal operator is solved by the servers to update the model in Algorithm 3. In the context of sketching (Sec. 5.3), almost all operations occur on the server side.

### 3.4 Asynchronous Tasks and Dependency

A tasks is issued by a remote procedure call. It can be a `push` or a `pull` that a worker issues to servers. It can also be a user-defined function that the scheduler issues to any node. Tasks may include any number of subtasks. For example, the task `WorkerIterate` in Algorithm 1 contains one `push` and one `pull`.

Tasks are executed asynchronously: the caller can perform further computation immediately after issuing a task.

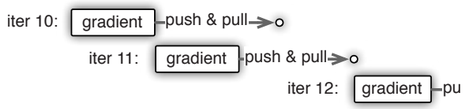


Figure 5: Iteration 12 depends on 11, while 10 and 11 are independent, thus allowing asynchronous processing.

The caller marks a task as finished only once it receives the callee's reply. A reply could be the function return of a user-defined function, the (key,value) pairs requested by the `pull`, or an empty acknowledgement. The callee marks a task as finished only if the call of the task is returned and all subtasks issued by this call are finished.

By default, callees execute tasks in parallel, for best performance. A caller that wishes to serialize task execution can place an *execute-after-finished* dependency between tasks. Figure 5 depicts three example iterations of `WorkerIterate`. Iterations 10 and 11 are independent, but 12 depends on 11. The callee therefore begins iteration 11 immediately after the local gradients are computed in iteration 10. Iteration 12, however, is postponed until the `pull` of 11 finishes.

Task dependencies help implement algorithm logic. For example, the aggregation logic in `ServerIterate` of Algorithm 1 updates the weight $w$ only after all worker gradients have been aggregated. This can be implemented by having the updating task depend on the push tasks of all workers. The second important use of dependencies is to support the flexible consistency models described next.

### 3.5 Flexible Consistency

Independent tasks improve system efficiency via parallelizing the use of CPU, disk and network bandwidth. However, this may lead to data inconsistency between nodes. In the diagram above, the worker $r$ starts iteration 11 before $w^{(11)}$ has been pulled back, so it uses the old $w_r^{(10)}$ in this iteration and thus obtains the same gradient as in iteration 10, namely $g_r^{(11)} = g_r^{(10)}$. This inconsistency potentially slows down the convergence progress of Algorithm 1. However, some algorithms may be less sensitive to this type of inconsistency. For example, only a segment of $w$ is updated each time in Algorithm 3. Hence, starting iteration 11 without waiting for 10 causes only a part of $w$ to be inconsistent.

The best trade-off between system efficiency and algorithm convergence rate usually depends on a variety of factors, including the algorithm's sensitivity to data inconsistency, feature correlation in training data, and capacity



### 3.2 范围推送与拉取

数据通过推和拉操作在节点之间发送。在算法1中，每个工作节点将其整个局部梯度推送到服务器，然后拉取更新的权重。算法3中描述的更高级的算法使用相同的模式，只是每次只通信一组键。

参数服务器通过支持基于范围的推送与拉取，优化了这些更新，既方便了程序员，也提高了计算和网络带宽效率。如果 $\mathcal{R}$ 是一个键范围，那么w.push($\mathcal{R}$, dest)会将键范围$\mathcal{R}$中的所有现有条目 $w$ 发送到目标位置，目标位置可以是特定节点，也可以是节点组（如服务器组）。类似地，w.pull($\mathcal{R}$,dest)会从目标位置读取键范围 $\mathcal{R}$ 中的所有现有条目 $w$ 。如果我们设置 $\mathcal{R}$ 为整个键范围，那么整个向量 $w$ 将被通信。如果我们设置 $\mathcal{R}$ 包含单个键，那么只会有单个条目被发送。

这个接口可以扩展为通信任何与 $w$共享相同键的本地数据结构。例如，在算法1中，一个工作节点将其临时本地梯度 $g$ 推送到参数服务器进行聚合。一个选项是使 $g$ 全局共享。但是请注意， $g$与工作节点的工作集 $w$共享键。因此，程序员可以使用w.push($\mathcal{R}$, g, dest) 来推送本地梯度，以节省内存，并享受以下几节讨论的优化。

### 3.3 服务器上的用户自定义函数

除了从工作者那里聚合数据，服务器节点还可以执行用户自定义函数。这很有益，因为服务器节点通常对共享参数有更完整或更及时的信息。在算法1中，服务器节点评估正则化器Ω的次梯度以更新 $w$。同时，服务器通过解决算法3来更新模型一个更复杂的近端算子。在草图（第5.3节）的上下文中，几乎所有操作都在服务器端进行。

### 3.4 异步任务和依赖关系

一个任务是由远程过程调用发起的。它可以是工作者向服务器发起的推送或拉取。它也可以是调度程序向任何节点发出的用户自定义函数。任务可以包含任意数量的子任务。例如，算法1中的WorkerIterate任务包含一个推送和一个拉取。

任务异步执行：调用者可以在发出任务后立即进行进一步计算。

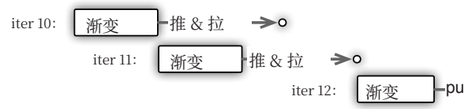


图5：迭代12依赖于11，而10和11是独立的，因此允许异步处理。

调用者只有在收到被调用者的回复后才会标记任务为完成。回复可能是用户定义函数的函数返回值、拉取请求的（键，值）对，或一个空的确认。被调用者只有在任务调用返回且该调用发出的所有子任务都完成后才会标记任务为完成。

默认情况下，被调用者并行执行任务，以获得最佳性能。希望串行化任务执行的调用者可以在任务之间放置一个“执行完成后依赖”关系。图5描绘了WorkerIterate的三个示例迭代。迭代10和11是独立的，但12依赖于11。因此，被调用者在迭代10中计算本地梯度后立即开始迭代11。然而，迭代12被推迟，直到11的拉取完成。

任务依赖关系有助于实现算法逻辑。例如，算法1中的ServerIterate的聚合逻辑仅在所有工作者梯度聚合后更新权重 $w$ 。这可以通过让更新任务依赖于所有工作者的推送任务来实现。依赖关系的第二个重要用途是支持接下来描述的灵活一致性模型。

### 3.5 弹性一致性

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。然而，这可能导致节点之间的数据不一致。在上面的图中，worker $r$ 在 $w^{(11)}$ 被拉回之前就开始了第 11 次迭代，因此在这次迭代中使用了旧的 $w_r^{(10)}$，从而获得了与第 10 次迭代相同的梯度，即 $g_r^{(11)} = g_r^{(10)}$。这种不一致性可能会减慢算法1的收敛进度。然而，某些算法可能对该类型的不一致性不太敏感。例如，在算法3中，每次只更新 $w$ 的一部分。因此，在不等待第10次迭代的情况下开始第11次迭代，只会导致 $w$ 的一部分不一致。

系统效率与算法收敛速度之间的最佳权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容量

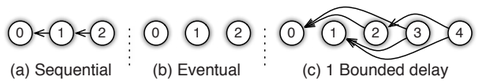


Figure 6: Directed acyclic graphs for different consistency models. The size of the DAG increases with the delay.

difference of hardware components. Instead of forcing the user to adopt one particular dependency that may be ill-suited to the problem, the parameter server gives the algorithm designer flexibility in defining consistency models. This is a substantial difference to other machine learning systems.

We show three different models that can be implemented by task dependency. Their associated directed acyclic graphs are given in Figure 6.

**Sequential** In sequential consistency, all tasks are executed one by one. The next task can be started only if the previous one has finished. It produces results identical to the single-thread implementation, and also named Bulk Synchronous Processing.

**Eventual** Eventual consistency is the opposite: all tasks may be started simultaneously. For instance, [43] describes such a system. However, this is only recommendable if the underlying algorithms are robust with regard to delays.

**Bounded Delay** When a maximal delay time $\tau$ is set, a new task will be blocked until all previous tasks $\tau$ times ago have been finished. Algorithm 3 uses such a model. This model provides more flexible controls than the previous two: $\tau = 0$ is the sequential consistency model, and an infinite delay $\tau = \infty$ becomes the eventual consistency model.

Note that the dependency graphs may be dynamic. For instance the scheduler may increase or decrease the maximal delay according to the runtime progress to balance system efficiency and convergence of the underlying optimization algorithm. In this case the caller traverses the DAG. If the graph is static, the caller can send all tasks with the DAG to the callee to reduce synchronization cost.

### 3.6 User-defined Filters

Complementary to a scheduler-based flow control, the parameter server supports user-defined filters to selectively synchronize individual (key,value) pairs, allowing fine-grained control of data consistency within a task. The insight is that the optimization algorithm itself usually possesses information on which parameters are most useful for synchronization. One example is the *significantly modified* filter, which only pushes entries that have changed by more than a threshold since their last synchronization. In Section 5.1, we discuss another filter named *KKT* which takes advantage of the optimality condition of the optimization problem: a worker only pushes gradients that are likely to affect the weights on the servers.

**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**

## 4 Implementation

The servers store the parameters (key-value pairs) using consistent hashing [45] (Sec. 4.3). For fault tolerance, entries are replicated using chain replication [47] (Sec. 4.4). Different from prior (key,value) systems, the parameter server is optimized for *range based communication* with compression on both data (Sec. 4.2) and range based vector clocks (Sec. 4.1).

### 4.1 Vector Clock

Given the potentially complex task dependency graph and the need for fast recovery, each (key,value) pair is associated with a vector clock [30, 15], which records the time of each individual node on this (key,value) pair. Vector clocks are convenient, e.g., for tracking aggregation status or rejecting doubly sent data. However, a naive implementation of the vector clock requires $O(nm)$ space to handle $n$ nodes and $m$ parameters. With thousands of nodes and billions of parameters, this is infeasible in terms of memory and bandwidth.

Fortunately, many parameters hare the same timestamp as a result of the range-based communication pattern of the parameter server: If a node pushes the parameters in a range, then the timestamps of the parameters associated with the node are likely the same. Therefore, they can be compressed into a single range vector clock. More specifically, assume that $\mathrm{vc}_i(k)$ is the time of key $k$ for node $i$. Given a key range $\mathcal{R}$, the ranged vector clock $\mathrm{vc}_i(\mathcal{R}) = t$ means for any key $k \in \mathcal{R}$, $\mathrm{vc}_i(k) = t$.

Initially, there is only one range vector clock for each node $i$. It covers the entire parameter key space as its




图6：不同一致性模型的有向无环图。DAG的大小随延迟增加而增大。

硬件组件的差异。参数服务器不会强迫用户采用可能不适合问题的特定依赖关系，而是为算法设计者定义一致性模型提供了灵活性。这是与其他机器学习系统的一个重大区别。

我们展示了三种可以通过任务依赖实现的模型。它们的关联有向无环图如图6所示。

顺序一致性 在顺序一致性中，所有任务都是逐一执行的。下一个任务只有在前一个任务完成后才能开始。它产生的结果与单线程实现相同，也称为批量同步处理。

最终一致性 最终一致性则相反：所有任务可以同时开始。例如， [43]描述了这样的系统。然而，只有当底层算法在延迟方面足够健壮时，才推荐使用这种方法。

有界延迟 当设置最大延迟时间 $\tau$ 时，新任务将被阻塞，直到所有先前任务 $\tau$时间之前已经完成。算法3使用此类模型。与前面的两种模型相比，该模型提供了更灵活的控制： $\tau = 0$是顺序一致性模型，而无限延迟 $\tau = \infty$ 则成为最终一致性模型。

请注意，依赖图可能是动态的。例如，调度器可能会根据运行时进度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性。在这种情况下，调用者遍历DAG。如果图是静态的，调用者可以将带有DAG的所有任务发送给被调用者，以减少同步成本。

### 3.6 用户定义的过滤器

与基于调度的流控制互补的是，参数服务器支持用户自定义的过滤器，用于选择性地同步单个（键，值）对，从而实现对任务内数据一致性的细粒度控制。其核心思想是，优化算法本身通常拥有关于哪些参数最需要同步的信息。

**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**

一个例子是经过显著修改的过滤器，它只推送自上次同步以来变化超过阈值的条目。在5.1节中，我们讨论了另一个名为KKT的过滤器，该过滤器利用了优化问题的最优条件：工作者只推送可能影响服务器权重的梯度。

## 4 实现

服务器使用一致性哈希 [45] （第4.3节）存储参数（键值对）。为了容错性，条目使用链式复制 [47] （第4.4节）进行复制。与先前的（键，值）系统不同，参数服务器针对基于范围的通信进行了优化，并在数据（第4.2节）和基于范围的向量时钟（第4.1节）上使用压缩。

### 4.1 向量时钟

鉴于任务依赖图可能很复杂且需要快速恢复，每个（键，值）对都与一个向量时钟 [30, 15], 相关联，该时钟记录此（键，值）对上每个节点的每个节点的时间。向量时钟很方便，例如，用于跟踪聚合状态或拒绝重复发送的数据。然而，向量时钟的朴素实现需要 $O(nm)$ 空间来处理 $n$ 节点和 $m$ 参数。对于数千个节点和数十亿个参数来说，这在内存和带宽方面是不可行的。

幸运的是，许多参数由于参数服务器的基于范围的通信模式，其时间戳与结果相同：如果一个节点推送范围内的参数，那么与该节点关联的参数的时间戳很可能相同。因此，它们可以被压缩成一个单一的范围向量时钟。更具体地说，假设$\mathrm{vc}_i(k)$是节点 $i$的键 $k$ 的时间。给定一个键范围 $\mathcal{R}$，范围向量时钟 $\mathrm{vc}_i(\mathcal{R}) = t$意味着对于任何键 $k \in \mathcal{R}$，$\mathrm{vc}_i(k) = t$。

最初，每个节点 $i$只有一个范围向量时钟。它覆盖了整个参数键空间作为其

range with 0 as its initial timestamp. Each range set may split the range and create at most 3 new vector clocks (see Algorithm 2). Let $k$ be the total number of unique ranges communicated by the algorithm, then there are at most $\mathcal{O}(mk)$ vector clocks, where $m$ is the number of nodes. $k$ is typically much smaller than the total number of parameters. This significantly reduces the space required for range vector clocks.[3]

### 4.2 Messages

Nodes may send messages to individual nodes or node groups. A message consists of a list of (key,value) pairs in the key range $\mathcal{R}$ and the associated range vector clock:

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

This is the basic communication format of the parameter server not only for shared parameters but also for tasks. For the latter, a (key,value) pair might assume the form (task ID, arguments or return results).

Messages may carry a subset of all available keys within range $\mathcal{R}$. The missing keys are assigned the same timestamp without changing their values. A message can be split by the key range. This happens when a worker sends a message to the whole server group, or when the key assignment of the receiver node has changed. By doing so, we partition the (key,value) lists and split the range vector clock similar to Algorithm 2.

Because machine learning problems typically require high bandwidth, message compression is desirable. Training data often remains unchanged between iterations. A worker might send the same key lists again. Hence it is desirable for the receiving node to cache the key lists. Later, the sender only needs to send a hash of the list rather than the list itself. Values, in turn, may contain many zero entries. For example, a large portion of parameters remain unchanged in sparse logistic regression, as evaluated in Section 5.1. Likewise, a user-defined filter may also zero out a large fraction of the values (see Figure 12). Hence we need only send nonzero (key,value) pairs. We use the fast Snappy compression library [21] to compress messages, effectively removing the zeros. Note that key-caching and value-compression can be used jointly.

### 4.3 Consistent Hashing

The parameter server partitions keys much as a conventional distributed hash table does [8, 41]: keys and server node IDs are both inserted into the hash ring (Figure 7). Each server node manages the key range starting with its insertion point to the next point by other nodes in the counter-clockwise direction. This node is called the master of this key range. A physical server is often represented in the ring via multiple "virtual" servers to improve load balancing and recovery.

We simplify the management by using a direct-mapped DHT design. The server manager handles the ring management. All other nodes cache the key partition locally. This way they can determine directly which server is responsible for a key range, and are notified of any changes.

### 4.4 Replication and Consistency

Each server node stores a replica of the $k$ counterclockwise neighbor key ranges relative to the one it owns. We refer to nodes holding copies as slaves of the appropriate key range. The above diagram shows an example with $k = 2$, where server 1 replicates the key ranges owned by server 2 and server 3.

Worker nodes communicate with the master of a key range for both `push` and `pull`. Any modification on the master is copied with its timestamp to the slaves. Modifications to data are pushed synchronously to the slaves. Figure 8 shows a case where worker 1 pushes $x$ into server 1, which invokes a user defined function $f$ to modify the shared data. The push task is completed only once the data modification $f(x)$ is copied to the slave.

Naive replication potentially increases the network traffic by $k$ times. This is undesirable for many machine learning applications that depend on high network bandwidth. The parameter server framework permits an important optimization for many algorithms: replication after aggregation. Server nodes often aggregate data from the worker nodes, such as summing local gradients. Servers may therefore postpone replication until aggregation is complete. In the righthand side of the diagram, two workers push $x$ and $y$ to the server, respectively. The server first aggregates the push by $x + y$, then applies the modification $f(x+y)$, and finally performs the replication. With $n$ workers, replication uses only $k/n$ bandwidth. Often $k$ is a small constant, while $n$ is hundreds to thousands. While aggregation increases the delay of the task reply, it can be hidden by relaxed consistency conditions.

### 4.5 Server Management

To achieve fault tolerance and dynamic scaling we must support addition and removal of nodes. For convenience we refer to virtual servers below. The following steps happen when a server joins.

[3] Ranges can be also merged to reduce the number of fragments. However, in practice both $m$ and $k$ are small enough to be easily handled. We leave merging for future work.



范围，初始时间戳为 0。每个范围集合可以拆分范围，最多创建 3 个新的向量时钟（参见算法 2）。设 $k$ 为算法通信的唯一范围总数，那么最多有 $\mathcal{O}(mk)$ 个向量时钟，其中 $m$ 是节点数。$k$ 通常远小于参数总数。这显著减少了范围向量时钟所需的存储空间。[3]

### 4.2 消息

节点可以向单个节点或节点组发送消息。一条消息由一组(key,value)对组成，这些对的键位于范围 $\mathcal{R}$ 内，并附带相关的范围向量时钟：

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

这是参数服务器的基本通信格式，不仅用于共享参数，也用于任务。对于后者，(key,value)对可能采用(taskID,参数或返回结果)的形式。

消息可以携带范围 $\mathcal{R}$内所有可用键的子集。缺失的键会被赋予相同的时戳，但值不会改变。消息可以按键范围进行拆分。这发生在工作节点向整个服务器组发送消息时，或者接收节点的键分配发生变化时。通过这种方式，我们将(key,value)列表分区，并像算法2一样拆分范围向量时钟。

由于机器学习问题通常需要高带宽，因此消息压缩是理想的。训练数据在迭代之间通常保持不变。一个工作节点可能会再次发送相同的键列表。因此，接收节点缓存键列表是理想的。稍后，发送者只需发送列表的哈希值，而不是列表本身。值反过来可能包含许多零条目。例如，在稀疏逻辑回归中，如第5.1节所述，大部分参数保持不变。同样，用户定义的过滤器也可能将很大一部分值置零（见图12）。因此，我们只需发送非零（键，值）对。我们使用快速Snappy压缩库 [21] 压缩消息，有效地移除零值。请注意，键缓存和值压缩可以联合使用。

### 4.3 一致性哈希

参数服务器对键的分区与常规分布式哈希表类似 [8, 41]：键和服务器参数服务器对键的分区方式与传统的分布式哈希表类似{v1}：键和服务器节点 ID 都会被插入到哈希环中（图 7）。每个服务器节点管理从其插入点开始，按逆时针方向到其他节点插入点的键范围。该节点被称为此键范围的“主节点”。物理服务器通常通过多个“虚拟”服务器在环中表示，以改善负载均衡和恢复。

我们通过使用直接映射的DHT设计简化了管理。服务器管理器负责环的管理。所有其他节点在本地缓存键分区。这样它们可以直接确定哪个服务器负责某个键范围，并会收到任何变更的通知。

### 4.4 复制与一致性

每个服务器节点存储其拥有的逆时针相邻键范围的副本。我们称持有副本的节点为相应键范围的从属节点。上面的图表展示了一个使用$k = 2$的示例，其中服务器1复制了服务器2和服务器3拥有的键范围。

工作节点与键范围的主节点进行推和拉通信。主节点上的任何修改都会附带时间戳复制到从属节点。对数据的修改会同步推送到从属节点。图8展示了一个工作1将 $x$ 推送到服务器1的案例，这会调用用户定义的函数 $f$ 来修改共享数据。只有当数据修改 $f(x)$复制到从属节点后，推任务才会完成。

朴素复制可能会使网络流量增加 $k$ 倍。这对许多依赖高网络带宽的机器学习应用来说是不利的。参数服务器框架为许多算法提供了一种重要的优化：聚合后复制。服务器节点通常会从工作节点聚合数据，例如对本地梯度求和。因此，服务器可以等到聚合完成后才进行复制。在图的右侧，两个工作者分别将 $x$ 和 $y$ 推送到服务器。服务器首先通过 $x + y$ 进行聚合，然后应用修改 $f(x+y)$，最后执行复制。对于 $n$个工作者，复制仅使用 $k/n$ 带宽。通常 $k$ 是一个小的常数，而 $n$ 是几百到几千。虽然聚合会增加任务回复的延迟，但可以通过宽松一致性条件来隐藏。

### 4.5 服务器管理

要实现容错性和动态扩展，我们必须支持节点的添加和移除。为方便起见，下文将虚拟服务器简称为虚拟机。当服务器加入时，会发生以下步骤。

[3] 范围也可以合并以减少片段数量。然而，在实践中，$m$ 和 $k$ 都足够小，可以轻松处理。我们将合并留待未来工作。

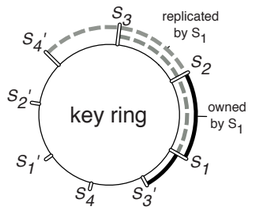


Figure 7: Server node layout.

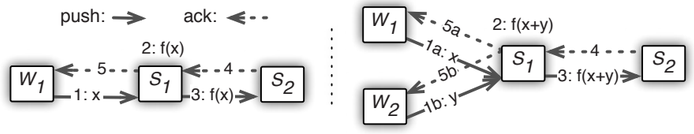


Figure 8: Replica generation. Left: single worker. Right: multiple workers updating values simultaneously.

1. The server manager assigns the new node a key range to serve as master. This may cause another key range to split or be removed from a terminated node.
2. The node fetches the range of data to maintains as master and $k$ additional ranges to keep as slave.
3. The server manager broadcasts the node changes. The recipients of the message may shrink their own data based on key ranges they no longer hold and to resubmit unfinished tasks to the new node.

Fetching the data in the range $\mathcal{R}$ from some node $S$ proceeds in two stages, similar to the Ouroboros protocol [38]. First $S$ pre-copies all (key,value) pairs in the range together with the associated vector clocks. This may cause a range vector clock to split similar to Algorithm 2. If the new node fails at this stage, $S$ remains unchanged. At the second stage $S$ no longer accepts messages affecting the key range $\mathcal{R}$ by dropping the messages without executing and replying. At the same time, $S$ sends the new node all changes that occurred in $\mathcal{R}$ during the pre-copy stage.

On receiving the node change message a node $N$ first checks if it also maintains the key range $\mathcal{R}$. If true and if this key range is no longer to be maintained by $N$, it deletes all associated (key,value) pairs and vector clocks in $\mathcal{R}$. Next, $N$ scans all outgoing messages that have not received replies yet. If a key range intersects with $\mathcal{R}$, then the message will be split and resent.

Due to delays, failures, and lost acknowledgements $N$ may send messages twice. Due to the use of vector clocks both the original recipient and the new node are able to reject this message and it does not affect correctness.

The departure of a server node (voluntary or due to failure) is similar to a join. The server manager tasks a new node with taking the key range of the leaving node. The server manager detects node failure by a heartbeat signal. Integration with a cluster resource manager such as Yarn [17] or Mesos [23] is left for future work.

### 4.6 Worker Management

Adding a new worker node $W$ is similar but simpler than adding a new server node:

1. The task scheduler assigns $W$ a range of data.
2. This node loads the range of training data from a network file system or existing workers. Training data is often read-only, so there is no two-phase fetch. Next, $W$ pulls the shared parameters from servers.
3. The task scheduler broadcasts the change, possibly causing other workers to free some training data.

When a worker departs, the task scheduler may start a replacement. We give the algorithm designer the option to control recovery for two reasons: If the training data is huge, recovering a worker node be may more expensive than recovering a server node. Second, losing a small amount of training data during optimization typically affects the model only a little. Hence the algorithm designer may prefer to continue without replacing a failed worker. It may even be desirable to terminate the slowest workers.

## 5 Evaluation

We evaluate our parameter server based on the use cases of Section 2 — Sparse Logistic Regression and Latent Dirichlet Allocation. We also show results of sketching to illustrate the generality of our framework. The experiments were run on clusters in two (different) large internet companies and a university research cluster to demonstrate the versatility of our approach.

### 5.1 Sparse Logistic Regression

**Problem and Data:** Sparse logistic regression is one of the most popular algorithms for large scale risk minimization [9]. It combines the logistic loss[4] with the $\ell_1$

[4] $\ell(x_i, y_i, w) = \log(1 + \exp(-y_i \langle x_i, w \rangle))$



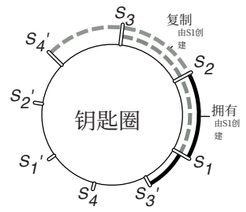


图7：服务器节点布局。

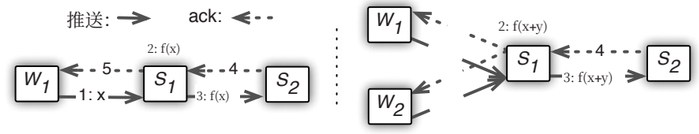


图8：副本生成。左：单个工作者。右：多个工作者同时更新值。

1. 服务器管理器为新节点分配一个键范围作为主节点。这可能导致另一个键范围从已终止的节点中分裂或被移除。

2. 节点获取要维护为主节点的数据范围和 $k$ 额外范围作为从节点。

3. 服务器管理器广播节点变更。接收消息的节点可能会根据不再持有的键范围缩小自己的数据，并将未完成的任务重新提交给新节点。

从某个节点 $S$获取范围 $\mathcal{R}$ 中的数据分为两个阶段，类似于Ouroboros协议 [38]。首先 $S$ 预先复制范围内的所有(key,value)对及其相关的向量时钟。这可能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， $S$ 保持不变。在第二阶段 $S$ 不再接受影响键范围 $\mathcal{R}$ 的消息，通过丢弃不执行和回复的消息来实现。与此同时， $S$ 将预复制阶段 $\mathcal{R}$ 期间发生的所有变更发送给新节点。

在接收到节点变更消息后，节点 $N$ 首先检查它是否也维护键范围 $\mathcal{R}$。如果为真，并且如果此键范围不再由 $N$ 维护，则它删除所有相关的 (键,值) 对和向量时钟在 $\mathcal{R}$。接下来， $N$ 扫描所有尚未收到回复的出站消息。如果一个键范围与 $\mathcal{R}$ 相交，那么消息将被拆分并重发。

由于延迟、故障和丢失的确认， $N$ 可能会发送两次消息。由于使用向量时钟，原始接收者和新节点都能够拒绝此消息，并且它不会影响正确性。

服务器节点的退出（自愿退出或因故障）类似于加入。服务器管理器会指派一个新节点来接管退出节点的键范围。服务器管理器通过心跳信号检测节点故障。与 Yarn [17] 或 Mesos [23] 等集群资源管理器的集成留待未来工作。

### 4.6 工作者管理

添加一个新的工作节点 $W$ 与添加一个新服务器节点类似但更简单：

1. 任务调度器分配 $W$ 一组数据。
2. 此节点从网络文件系统或现有工作者加载训练数据范围。训练数据通常是只读的，所以没有两阶段获取。接下来，$W$ 从服务器拉取共享参数。
3. 任务调度器广播变更，可能导致其他工作者释放部分训练数据。

当工作者离开时，任务调度器可能会启动替换。我们给算法设计者提供控制恢复的选项，原因有两个：如果训练数据量巨大，恢复一个工作者节点可能比恢复一个服务器节点更昂贵。其次，在优化过程中丢失少量训练数据通常只会对模型产生轻微影响。因此，算法设计者可能倾向于不替换故障工作者。甚至可能希望终止最慢的工作者。

## 5 评估

我们根据第2节的用例——稀疏逻辑回归和潜在狄利克雷分配，评估我们的参数服务器。我们还展示了草稿结果，以说明我们框架的通用性。实验在两家（不同）大型互联网公司的集群和一所大学的研究集群上运行，以展示我们方法的通用性。

### 5.1 稀疏逻辑回归

问题与数据：稀疏逻辑回归是用于大规模风险最小化的最流行算法之一 [9]。它结合了逻辑损失[4] 与 $\ell_1$

[4] $\ell(x_i, y_i, w) = \log(1 + \exp(-y_i \langle x_i, w \rangle))$

**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 $\|x\|_H^2 = x^T H x$

| | Method | Consistency | LOC |
|---|---|---|---|
| System A | L-BFGS | Sequential | 10,000 |
| System B | Block PG | Sequential | 30,000 |
| Parameter Server | Block PG | Bounded Delay KKT Filter | 300 |

Table 3: Systems evaluated.

regularizer[5] of Section 2.2. The latter biases a compact solution with a large portion of 0 value entries. The non-smoothness of this regularizer, however, makes learning more difficult.

We collected an ad click prediction dataset with 170 billion examples and 65 billion unique features. This dataset is 636 TB uncompressed (141 TB compressed). We ran the parameter server on 1000 machines, each with 16 physical cores, 192GB DRAM, and connected by 10 Gb Ethernet. 800 machines acted as workers, and 200 were parameter servers. The cluster was in concurrent use by other (unrelated) tasks during operation.

**Algorithm:** We used a state-of-the-art distributed regression algorithm (Algorithm 3, [31, 32]). It differs from the simpler variant described earlier in four ways: First, only a block of parameters is updated in an iteration. Second, the workers compute both gradients and the diagonal part of the second derivative on this block. Third, the parameter servers themselves must perform complex computation: the servers update the model by solving a *proximal operator* based on the aggregated local gradients. Fourth, we use a bounded-delay model over iterations and use a "KKT" filter to suppress transmission of parts of the generated gradient update that are small enough that their effect is likely to be negligible.[6]

[5] $\Omega(w) = \sum_{i=1}^n |w_i|$

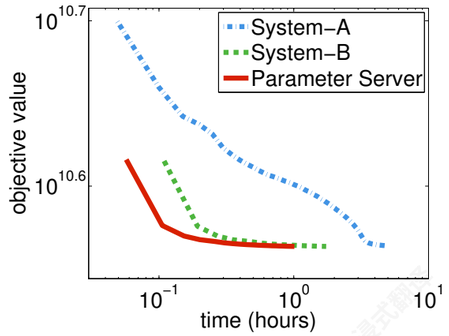


Figure 9: Convergence of sparse logistic regression. The goal is to minimize the objective rapidly.

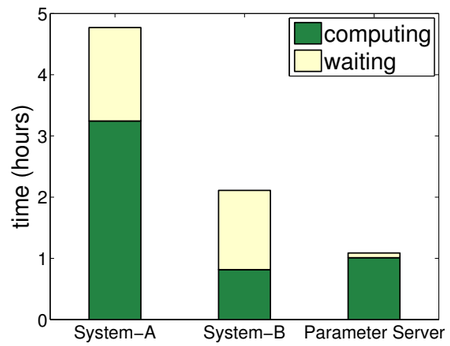


Figure 10: Time per worker spent on computation and waiting during sparse logistic regression.

To the best of our knowledge, no open source system can scale sparse logistic regression to the scale described in this paper.[7] We compare the parameter server with two special-purpose systems, named System A and B, devel-

[6] A user-defined Karush-Kuhn-Tucker (KKT) filter [26]. Feature $k$ is filtered if $w_k = 0$ and $|\hat{g}_k| \leq \Delta$. Here $\hat{g}_k$ is an estimate of the global gradient based on the worker's local information and $\Delta > 0$ is a user-defined parameter.

[7] Graphlab provides only a multi-threaded, single machine implementation, while Petuum, Mlbase and REEF do not support sparse logistic regression. We confirmed this with the authors as per 4/2014.



**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 $\|x\|_H^2 = x^T H x$

| | Method | Consistency | LOC |
|---|---|---|---|
| System A | L-BFGS | Sequential | 10,000 |
| System B | Block PG | Sequential | 30,000 |
| Parameter Server | Block PG | Bounded Delay KKT Filter | 300 |

表3：评估的系统

第2.2节的正则化器[5]。后者使具有大量0值条目的紧凑解产生偏差。然而，这种正则化器的非光滑性使得学习更加困难。

我们收集了一个包含1700亿个样本和650亿个唯一特征的广告点击预测数据集。该数据集未压缩时为636TB（压缩后为141TB）。我们在1000台机器上运行参数服务器，每台机器拥有16个物理核心、192GB DRAM，并通过10Gb以太网连接。其中800台机器作为工作者，200台作为参数服务器。该集群在运行期间被其他（无关）任务并发使用。

算法：我们使用了一种最先进的分布式回归算法（算法 3, [31, 32]）。它与之前描述的简单变体有四个不同之处：首先，在一次迭代中只更新一个参数块。其次，工作者在这个块上计算梯度以及二阶导数的对角部分。第三，参数服务器本身必须执行复杂的计算：服务器通过求解基于聚合局部梯度的近似算子来更新模型。第四，我们对迭代使用有界延迟模型，并使用“KKT”过滤器来抑制那些影响可能可以忽略不计的梯度更新部分的传输. 3, [31, 32]

[5] $\Omega(w) = \sum_{i=1}^n |w_i|$

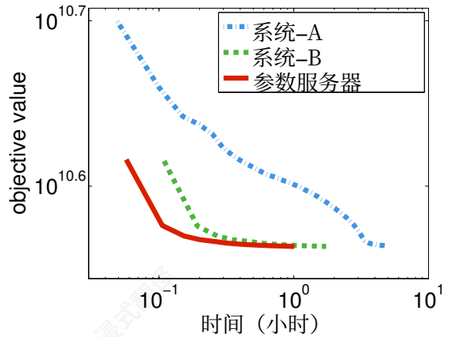


图9：稀疏逻辑回归的收敛性。目标是在快速最小化目标函数。

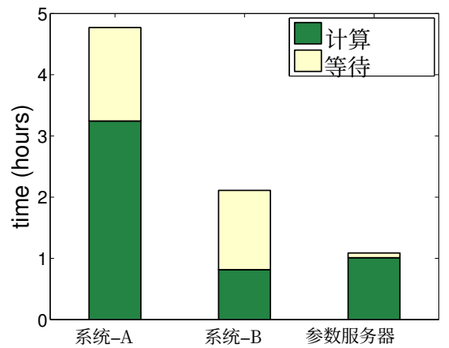


图 10：稀疏逻辑回归中每个工作节点花费在计算和等待上的时间。

计算：服务器通过求解基于聚合局部梯度的近似算子来更新模型。第四，我们对迭代使用有界延迟模型，并使用“KKT”过滤器来抑制那些影响可能可以忽略不计的梯度更新部分的传输.[6]

据我们所知，没有开源系统能够将稀疏逻辑回归扩展到本文所述的规模.[7] 我们将参数服务器与两个专用系统进行比较，这两个系统名为系统A和系统B，由一家大型互联网公司开发。

[6] 一个用户定义的KKT(Karush-Kuhn-Tucker)过滤器 [26]。如果满足 $w_k = 0$和 $\hat{g}_k| \leq \Delta.$，则过滤特征 $k$。其中$\hat{g}_k$ 是基于工作节点局部信息估计的全局梯度， $\Delta > 0$ 是一个用户定义的参数。

[7] Graphlab仅提供多线程单机实现，而Petuum、Mlbase和REEF不支持稀疏逻辑回归。我们根据4/2014与作者确认了这一点。

oped by a large internet company.

Notably, both Systems A and B consist of more than 10K lines of code. The parameter server only requires 300 lines of code for the same functionality as System B.[8] The parameter server successfully moves most of the system complexity from the algorithmic implementation into a reusable generalized component.

**Results:** We first compare these three systems by running them to reach the same objective value. A better system achieves a lower objective in less time. Figure 9 shows the results: System B outperforms system A because it uses a better algorithm. The parameter server, in turn, outperforms System B while using the same algorithm. It does so because of the efficacy of reducing the network traffic and the relaxed consistency model.

Figure 10 shows that the relaxed consistency model substantially increases worker node utilization. Workers can begin processing the next block without waiting for the previous one to finish, hiding the delay otherwise imposed by barrier synchronization. Workers in System A are 32% idle, and in system B, they are 53% idle, while waiting for the barrier in each block. The parameter server reduces this cost to under 2%. This is not entirely free: the parameter server uses slightly more CPU than System B for two reasons. First, and less fundamentally, System B optimizes its gradient calculations by careful data preprocessing. Second, asynchronous updates with the parameter server require more iterations to achieve the same objective value. Due to the significantly reduced communication cost, the parameter server halves the total time.

Next we evaluate the reduction of network traffic by each system components. Figure 11 shows the results for servers and workers. As can be seen, allowing the senders and receivers to cache the keys can save near 50% traffic. This is because both key (`int64`) and value (`double`) are of the same size, and the key set is not changed during optimization. In addition, data compression is effective for compressing the values for both servers (>20x) and workers when applying the KKT filter (>6x). The reason is twofold. First, the $\ell_1$ regularizer encourages a sparse model ($w$), so that most of values pulled from servers are 0. Second, the KKT filter forces a large portion of gradients sending to servers to be 0. This can be seen more clearly in Figure 12, which shows that more than 93% unique features are filtered by the KKT filter.

Finally, we analyze the bounded delay consistency model. The time decomposition of workers to achieve the same convergence criteria under different maximum allowed delay ($\tau$) is shown in Figure 13. As expected, the waiting time decreases when the allowed delay increases. Workers are 50% idle when using the sequential consistency model ($\tau = 0$), while the idle rate is reduced to 1.7% when $\tau$ is set to be 16. However, the computing time increases nearly linearly with $\tau$. Because the data inconsistency slows convergence, more iterations are needed to achieve the same convergence criteria. As a result, $\tau = 8$ is the best trade-off between algorithm convergence and system performance.

[8] System B was developed by an author of this paper.

## 5.2 Latent Dirichlet Allocation

**Problem and Data:** To demonstrate the versatility of our approach, we applied the same parameter server architecture to the problem of modeling user interests based upon which domains appear in the URLs they click on in search results. We collected search log data containing 5 billion unique user identifiers and evaluated the model for the 5 million most frequently clicked domains in the result set. We ran the algorithm using 800 workers and 200 servers and 5000 workers and 1000 servers respectively. The machines had 10 physical cores, 128GB DRAM, and at least 10 Gb/s of network connectivity. We again shared the cluster with production jobs running concurrently.

**Algorithm:** We performed LDA using a combination of Stochastic Variational Methods [25], Collapsed Gibbs sampling [20] and distributed gradient descent. Here, gradients are aggregated asynchronously as they arrive from workers, along the lines of [1].

We divided the parameters in the model into local and global parameters. The local parameters (i.e. auxiliary metadata) are pertinent to a given user and they are streamed the from disk whenever we access a given user. The global parameters are shared among users and they are represented as (key,value) pairs to be stored using the parameter server. User data is sharded over workers. Each of them runs a set of computation threads to perform inference over its assigned users. We synchronize asynchronously to send and receive local updates to the server and receive new values of the global parameters.

To our knowledge, no other system (e.g., YahooLDA, Graphlab or Petuum) can handle this amount of data and model complexity for LDA, using up to 10 billion (5 million tokens and 2000 topics) shared parameters. The largest previously reported experiments [2] had under 100 million users active at any time, less than 100,000 tokens and under 1000 topics (2% the data, 1% the parameters).

**Results:** To evaluate the quality of the inference algorithm we monitor how rapidly the training log-likelihood



由一家大型互联网公司开发。

值得注意的是，系统A和系统B都包含超过10K行的代码。参数服务器仅需要300行代码即可实现与系统B相同的功能。[8] 参数服务器成功地将大部分系统复杂性从算法实现中转移到可重用的通用组件中。

结果：我们首先通过运行这三个系统以达到相同的目标值来比较它们。更好的系统能在更短的时间内实现更低的目标值。图9显示了结果：系统B比系统A表现更好，因为它使用了更好的算法。参数服务器则在使用相同算法的情况下优于系统B。这是因为它减少了网络流量和宽松一致性模型的有效性。

图10显示，宽松一致性模型显著提高了工作节点利用率。工作者可以开始处理下一个块，而无需等待前一个块完成，从而隐藏了屏障同步带来的延迟。系统A中的工作者有32%处于空闲状态，系统B中的工作者在等待每个块中的屏障时有53%处于空闲状态，而参数服务器将这一成本降低到不到2%。这并非完全免费：参数服务器比系统B稍微多使用一些CPU，原因有二。首先，系统B通过仔细的数据预处理优化了梯度计算，这不太根本。其次，与参数服务器的异步更新需要更多迭代才能达到相同的目标值。由于通信成本显著降低，参数服务器将总时间减半。

接下来，我们评估了每个系统组件对网络流量的减少。图11显示了服务器和工作者的结果。如图所示，允许发送者和接收者缓存密钥可以节省近50%的流量。这是因为密钥（int64）和值（double）的大小相同，并且在优化过程中密钥集没有改变。此外，在应用KKT过滤器（>6x）时，数据压缩对压缩服务器（>20x）和工作者的值非常有效。原因有两个。首先，$\ell_1$ 正则化器鼓励稀疏模型（$w$），因此从服务器检索的大多数值都是0。其次，KKT过滤器强制将大量梯度发送到服务器的部分设为0。这可以从图12中更清楚地看到，该图显示KKT过滤器过滤掉了超过93%的唯一特征。

最后，我们分析了有界延迟一致性模型。图13展示了在不同最大允许延迟($\tau$)下，工作者达到相同收敛标准所需的时间分解。如预期所示，

[8] 系统B是由本文的一位作者开发的。

允许延迟增加时，等待时间会减少。当使用顺序一致性模型 ($\tau = 0$)时，工作者空闲率为50%，而将 $\tau$ 设置为16时，空闲率降低至1.7%。然而，计算时间几乎随 $\tau$线性增加。由于数据不一致性会减缓收敛，需要更多迭代次数才能达到相同收敛标准。因此，$\tau = 8$是在算法收敛和系统性能之间的最佳权衡。

## 5.2 潜在狄利克雷分配

问题与数据：为了展示我们方法的通用性，我们将相同的参数服务器架构应用于建模用户兴趣的问题，即根据用户在搜索结果中点击的URL所属的领域来建模。我们收集了包含50亿个唯一用户标识符的搜索日志数据，并评估了结果集中5百万个最常被点击的领域。我们分别使用800个工作者和200个服务器、5000个工作者和1000个服务器运行了算法。这些机器拥有10个物理核心、128GB内存和至少10Gb/s的网络连接。我们再次与同时运行的生产任务共享了该集群。

算法：我们使用随机变分方法 [25], 折叠吉布斯采样 [20]和分布式梯度下降组合进行了LDA。在这里，梯度异步地聚合，就像从工作者那里到达一样，沿着 [1]的思路。

我们将模型中的参数分为局部参数和全局参数。局部参数（即辅助元数据）与特定用户相关，每次我们访问特定用户时都会从磁盘流式传输。全局参数在用户之间共享，并作为(key,value)对表示，使用参数服务器进行存储。用户数据在工作者之间分片。每个工作者运行一组计算线程，以对其分配的用户进行推理。我们异步同步以向服务器发送和接收局部更新，并接收全局参数的新值。

据我们所知，没有其他系统（例如YahooLDA、Graphlab或Petuum）能够处理LDA的如此大量数据和高模型复杂度，最多使用1000亿（500万个词元和2000个主题）共享参数。之前报道的最大实验 [2] 在任何时候都只有不到1亿活跃用户，词元数量不到10万个，主题数量不到1000个（数据量只有2%，参数量只有1%）。

结果：为评估推理算法的质量，需监控训练对数似然（衡量拟合优度）收敛的速度。

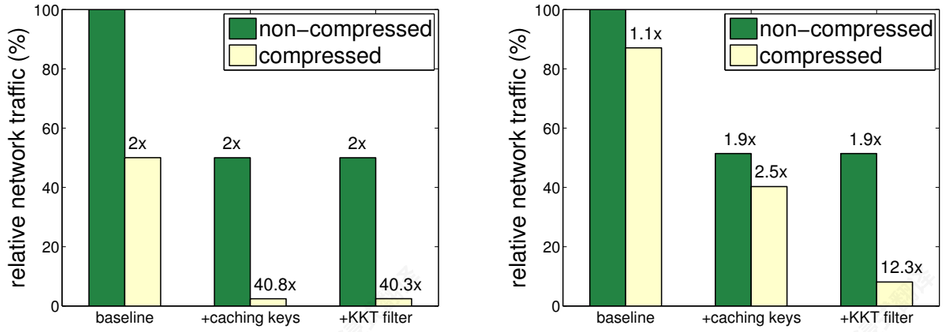


Figure 11: The savings of outgoing network traffic by different components. Left: per server. Right: per worker.

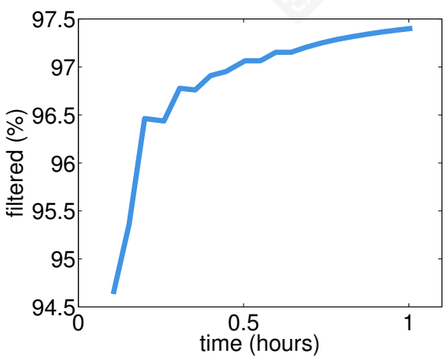


Figure 12: Unique features (keys) filtered by the KKT filter as optimization proceeds.

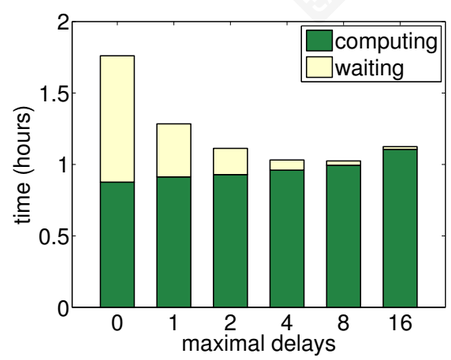


Figure 13: Time a worker spent to achieve the same convergence criteria by different maximal delays.

(measuring goodness of fit) converges. As can be seen in Figure 14, we observe an approximately 4x speedup in convergence when increasing the number of machines from 1000 to 6000. The stragglers observed in Figure 14 (leftmost) also illustrate the importance of having an architecture that can cope with performance variation across workers.

| Topic name | # Top urls |
|---|---|
| Programming | stackoverflow.com w3schools.com cplusplus.com github.com tutorialspoint.com jquery.com codeproject.com oracle.com qt-project.org bytes.com android.com mysql.com |
| Music | ultimate-guitar.com guitaretab.com 911tabs.com e-chords.com songsterr.com chordify.net musicnotes.com ukulele-tabs.com |
| Baby Related | babycenter.com whattoexpect.com babycentre.co.uk circleofmoms.com thebump.com parents.com momtastic.com parenting.com americanpregnancy.org kidshealth.org |
| Strength Training | bodybuilding.com muscleandfitness.com mensfitness.com menshealth.com t-nation.com livestrong.com muscleandstrength.com myfitnesspal.com elitefitness.com crossfit.com steroid.com gnc.com askmen.com |

Table 4: Example topics learned using LDA over the .5 billion dataset. Each topic represents a user interest

## 5.3 Sketches

**Problem and Data:** We include sketches as part of our evaluation as a test of generality, because they operate very differently from machine learning algorithms. They typically observe a large number of writes of events coming from a streaming data source [11, 5].

We evaluate the time required to insert a streaming log of pageviews into an approximate structure that can efficiently track pageview counts for a large collection of web pages. We use the Wikipedia (and other Wiki projects) page view statistics as benchmark. Each entry is an unique key of a webpage with the corresponding number of requests served in a hour. From 12/2007 to 1/2014, there are 300 billion entries for more than 100 million unique keys. We run the parameter server with 90 virtual server nodes on 15 machines of a research cluster [40] (each has



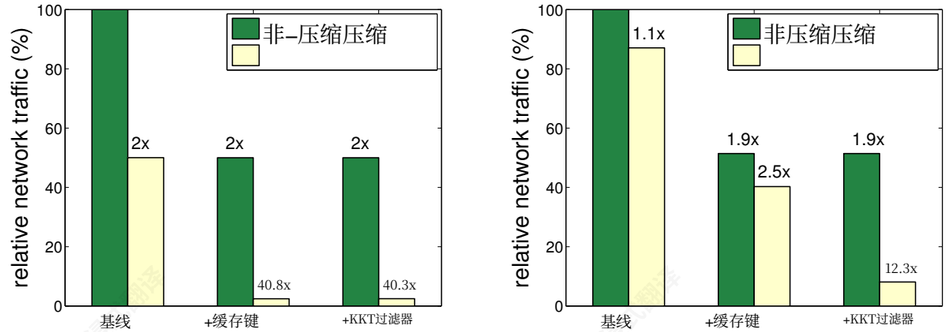


图11：不同组件节省的出站网络流量。左：每台服务器。右：每个工作节点。

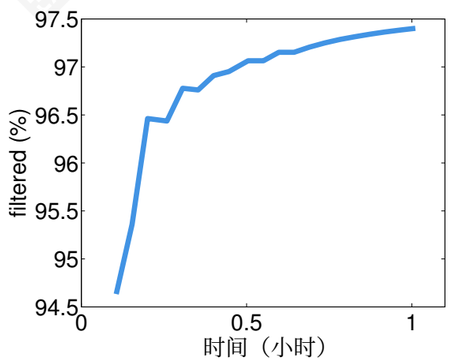


图12：KKTfilter在优化过程中过滤的独特特征（键）。

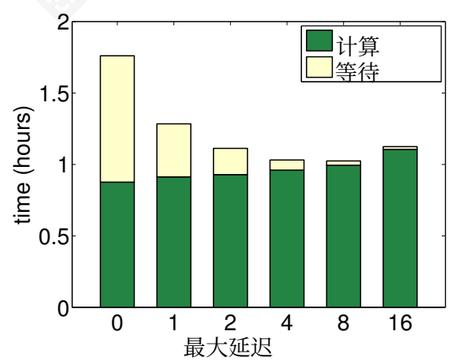


图13：不同最大延迟下，一个工人达到相同收敛标准所需的时间。

如图14所示，当机器数量从1000增加到6000时，我们观察到收敛速度提升了约4倍。图14最左侧显示的落后者（stragglers）也说明了应对工作者间性能变化（performance variation）的架构设计的重要性。

| 主题名称 | # 顶级网址 |
|---|---|
| 编程 | stackoverflow.com w3schools.com cplusplus.com github.com 教程-point.com jquery.com codeproject.com oracle.com qt-project.org bytes.com android.com mysql.com |
| 音乐 | ultimate-guitar.com guitaretab.com 911tabs.com e-chords.com 歌曲-sterr.com chordify.net musicnotes.com ukulele-tabs.com |
| 婴儿相关 | babycenter.com whattoexpect.com babycentre.co.uk circleofmoms.com thebump.com 父母.com 妈妈太棒.com 养育.com 美国怀孕-nancy.org kidshealth.org |
| 力量训练 | bodybuilding.com muscleandfitness.com mensfitness.com menshealth.com t-nation.com livestrong.com muscleandstrength.com myfitnesspal.com elitefitness.com crossfit.com steroid.com gnc.com askmen.com |

表4：示例使用LDA在5亿数据集上学习到的主题。每个主题代表一个用户兴趣

## 5.3 插图

问题和数据：我们将插图作为评估的一部分，因为它们作为通用性测试，与机器学习算法的操作方式非常不同。它们通常观察来自流数据源的事件的大量写入 [11, 5]。

我们评估了将页面浏览流式日志插入一个能够高效跟踪大量网页页面浏览次数的近似结构所需的时间。我们使用维基百科（及其他维基项目）的页面浏览统计数据作为基准。每条记录是一个网页的唯一键以及对应的小时内服务请求次数。从2007年12月至2014年1月，有3000亿条记录，涉及超过1亿个唯一键。我们在研究集群的15台机器上运行参数服务器 [40]（每台机器有

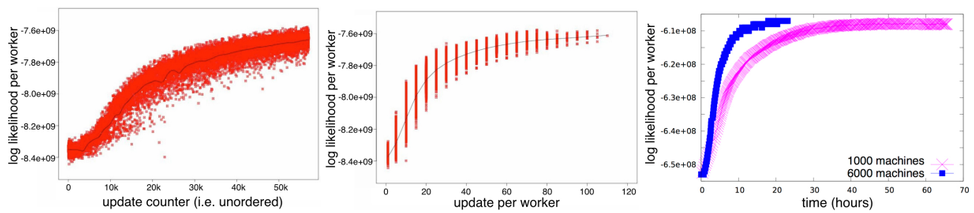


Figure 14: Left: Distribution over worker log-likelihoods as a function of time for 1000 machines and 5 billion users. Some of the low values are due to stragglers synchronizing slowly initially. Middle: the same distribution, stratified by the number of iterations. Right: convergence (time in 1000s) using 1000 and 6000 machines on 500M users.

**Algorithm 4** CountMin Sketch

**Init:** $M[i,j] = 0$ **for** $i \in \{1,\dots n\}$ **and** $j \in \{1,\dots k\}$.

**Insert**$(x)$

1: **for** $i = 1$ **to** $k$ **do**
2: $\quad M[i, \text{hash}(i,x)] \leftarrow M[i, \text{hash}(i,x)] + 1$

**Query**$(x)$

1: **return** $\min\{M[i, \text{hash}(i,x)] \text{ for } 1 \le i \le k\}$

64 cores and is connected by a 40Gb Ethernet).

**Algorithm:** Sketching algorithms efficiently store summaries of huge volumes of data so that approximate queries can be quickly answered. These algorithms are particularly important in streaming applications where data and queries arrive in real-time. Some of the highest-volume applications involve examples such as Cloudflare's DDoS-prevention service, which must analyze page requests across its entire content delivery service architecture to identify likely DDoS targets and attackers. The volume of data logged in such applications considerably exceeds the capacity of a single machine. While a conventional approach might be to shard a workload across a key-value cluster such as Redis, these systems typically do not allow the user-defined aggregation semantics needed to implement *approximate* aggregation.

Algorithm 4 gives a brief overview of the CountMin sketch [11]. By design, the result of a query is an *upper* bound on the number of observed keys $x$. Splitting keys into ranges automatically allows us to parallelize the sketch. Unlike the two previous applications, the workers simply dispatch updates to the appropriate servers.

**Results:** The system achieves very high insert rates, which are shown in Table 5. It performs well for two reasons: First, bulk communication reduces the communication cost. Second, message compression reduces the average (key,value) size to around 50 bits. Importantly, when we terminated a server node during the insertion, the parameter server was able to recover the failed node within 1 second, making our system well equipped for realtime.

| | |
|---|---|
| Peak inserts per second | 1.3 billion |
| Average inserts per second | 1.1 billion |
| Peak net bandwidth per machine | 4.37 GBit/s |
| Time to recover a failed node | 0.8 second |

Table 5: Results of distributed CountMin

## 6 Summary and Discussion

We described a parameter server framework to solve distributed machine learning problems. This framework is easy to use: Globally shared parameters can be used as local sparse vectors or matrices to perform linear algebra operations with local training data. It is efficient: All communication is asynchronous. Flexible consistentcy models are supported to balance the trade-off between system efficiency and fast algorithm convergence rate. Furthermore, it provides elastic scalability and fault tolerance, aiming for stable long term deployment. Finally, we show experiments for several challenging tasks on real datasets with billions of variables to demonstrate its efficiency. We believe that this third generation parameter server is an important building block for scalable machine learning. The codes are available at `parameterserver.org`.

**Acknowledgments:** This work was supported in part by gifts and/or machine time from Google, Amazon, Baidu, PRObE, and Microsoft; by NSF award 1409802; and by the Intel Science and Technology Center for Cloud Computing. We are grateful to our reviewers and colleagues for their comments on earlier versions of this paper.



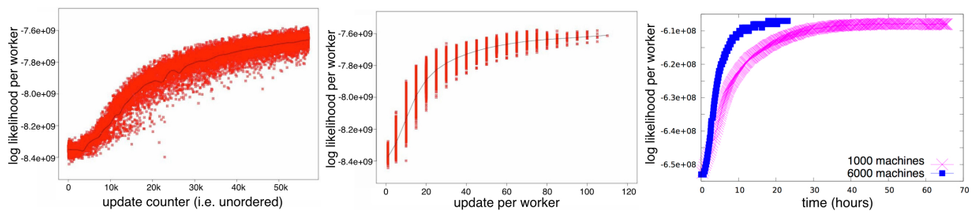


图14：左：1000台机器和50亿用户的工作者日志似然度随时间分布。部分低值是由于落后者初始同步缓慢造成的。中：相同分布，按迭代次数分层。右：使用1000台和6000台机器在5000万用户上的收敛情况（时间以千为单位）。

**Algorithm 4** CountMin Sketch

**Init:** $M[i,j] = 0$ **for** $i \in \{1,\dots n\}$ **and** $j \in \{1,\dots k\}$.

**Insert**$(x)$

1: **for** $i = 1$ **to** $k$ **do**
2: $\quad M[i, \text{hash}(i,x)] \leftarrow M[i, \text{hash}(i,x)] + 1$

**Query**$(x)$

1: **return** $\min\{M[i, \text{hash}(i,x)] \text{ for } 1 \le i \le k\}$

64个核心，并通过40Gb以太网连接）。

算法：Sketching算法高效地存储海量数据的摘要，以便快速回答近似查询。这些算法在流式应用中尤为重要，因为数据和查询实时到达。一些最高量的应用包括Cloudflare的DDoS防护服务等实例，该服务必须分析其整个内容分发服务架构中的页面请求，以识别可能的DDoS目标和攻击者。此类应用中记录的数据量远超单台机器的容量。虽然传统方法可能是将工作负载分片到Redis等键值集群，但这些系统通常不允许用户定义实现近似聚合所需的聚合语义。

算法4简要介绍了CountMin草图 [11]。按设计，查询的结果是对观测到的键数量的上限 $x$。将键拆分为范围可以自动实现并行化。与之前的两个应用不同，工作者只是将更新分派到相应的服务器。

| | |
|---|---|
| 每秒峰值插入量 | 13亿 |
| 每秒平均插入量 | 11亿 |
| 每台机器的峰值网络带宽 | 4.37 GBit/s |
| 恢复故障节点的耗时 | 0.8 second |

表5：分布式CountMin的结果

当插入时终止服务器节点期间，参数服务器能够在1秒内恢复故障节点，这使得我们的系统非常适合实时应用。

## 6 总结与讨论

我们描述了一个参数服务器框架，用于解决分布式机器学习问题。该框架易于使用：全局共享参数可以作为局部稀疏向量或矩阵，使用本地训练数据执行线性代数运算。它高效：所有通信都是异步的。支持灵活的一致性模型，以平衡系统效率与快速算法收敛率之间的权衡。此外，它提供弹性可扩展性和容错性，旨在实现稳定的长期部署。最后，我们在包含数十亿变量的真实数据集上展示了几个具有挑战性任务的实验，以证明其效率。我们相信，第三代参数服务器是可扩展机器学习的重要基础组件。代码可在parameterserver.org获取。

致谢：本研究部分由谷歌、亚马逊、百度、PRObE和微软的赠款和/或机时支持；由NSF奖项1409802支持；以及由英特尔云计算科学与技术中心支持。我们感谢我们的审稿人和同事对本文早期版本的评论。

结果：该系统实现了非常高的插入率，具体数据如表5所示。它表现良好，原因有两个：首先，批量通信降低了通信成本。其次，消息压缩将消息（键值）大小减少到约50比特。重要的是，在插入过程中我们终止了一个服务器节点时，参数服务器能够在1秒内恢复故障节点，这使得我们的系统非常适合实时应用。

## References


[1] A. Ahmed, M. Aly, J. Gonzalez, S. Narayanamurthy, and A. J. Smola. Scalable inference in latent variable models. In *Proceedings of The 5th ACM International Conference on Web Search and Data Mining (WSDM)*, 2012.

[2] A. Ahmed, Y. Low, M. Aly, V. Josifovski, and A. J. Smola. Scalable inference of dynamic user interests for behavioural targeting. In *Knowledge Discovery and Data Mining*, 2011.

[3] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, second edition, 1995.

[4] Apache Foundation. Mahout project, 2012. http://mahout.apache.org.

[5] R. Berinde, G. Cormode, P. Indyk, and M.J. Strauss. Space-optimal heavy hitters with strong error bounds. In J. Paredaens and J. Su, editors, *Proceedings of the Twenty-Eigth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS*, pages 157–166. ACM, 2009.

[6] C. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[7] D. Blei, A. Ng, and M. Jordan. Latent Dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, January 2003.

[8] J. Byers, J. Considine, and M. Mitzenmacher. Simple load balancing for distributed hash tables. In *Peer-to-peer systems II*, pages 80–87. Springer, 2003.

[9] K. Canini. Sibyl: A system for large scale supervised machine learning. *Technical Talk*, 2012.

[10] B.-G. Chun, T. Condie, C. Curino, C. Douglas, S. Matusevych, B. Myers, S. Narayanamurthy, R. Ramakrishnan, S. Rao, J. Rosen, R. Sears, and M. Weimer. Reef: Retainable evaluator execution framework. *Proceedings of the VLDB Endowment*, 6(12):1370–1373, 2013.

[11] G. Cormode and S. Muthukrishnan. Summarizing and mining skewed data streams. In *SDM*, 2005.

[12] W. Dai, J. Wei, X. Zheng, J. K. Kim, S. Lee, J. Yin, Q. Ho, and E. P. Xing. Petuum: A framework for iterative-convergent distributed ml. *arXiv preprint arXiv:1312.7651*, 2013.

[13] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. Le, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Ng. Large scale distributed deep networks. In *Neural Information Processing Systems*, 2012.

[14] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *CACM*, 51(1):107–113, 2008.

[15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In T. C. Bressoud and M. F. Kaashoek, editors, *Symposium on Operating Systems Principles*, pages 205–220. ACM, 2007.

[16] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14:18–32, 1988.

[17] The Apache Software Foundation. Apache hadoop nextgen mapreduce (yarn). http://hadoop.apache.org/.

[18] The Apache Software Foundation. Apache hadoop, 2009. http://hadoop.apache.org/core/.

[19] F. Girosi, M. Jones, and T. Poggio. Priors, stabilizers and basis functions: From regularization to radial, tensor and additive splines. A.I. Memo 1430, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1993.

[20] T.L. Griffiths and M. Steyvers. Finding scientific topics. *Proceedings of the National Academy of Sciences*, 101:5228–5235, 2004.

[21] S. H. Gunderson. Snappy: A fast compressor/decompressor. https://code.google.com/p/snappy/.

[22] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer, New York, 2 edition, 2009.

[23] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, pages 22–22, 2011.

[24] Q. Ho, J. Cipar, H. Cui, S. Lee, J. Kim, P. Gibbons, G. Gibson, G. Ganger, and E. Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *NIPS*, 2013.

[25] M. Hoffman, D. M. Blei, C. Wang, and J. Paisley. Stochastic variational inference. In *International Conference on Machine Learning*, 2012.

[26] W. Karush. Minima of functions of several variables with inequalities as side constraints. Master's thesis, Dept. of Mathematics, Univ. of Chicago, 1939.

[27] L. Kim. How many ads does Google serve in a day?, 2012. http://goo.gl/oIidXO.

[28] D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.

[29] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan. Mlbase: A distributed machine-learning system. In *CIDR*, 2013.

[30] L. Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[31] M. Li, D. G. Andersen, and A. J. Smola. Distributed delayed proximal gradient methods. In *NIPS Workshop on Optimization for Machine Learning*, 2013.




## 参考文献


[1] A. Ahmed, M. Aly, J. Gonzalez, S. Narayanamurthy, and A. J. Smola. Scalable inference in latent variable models. In *Proceedings of The 5th ACM International Conference on Web Search and Data Mining (WSDM)*, 2012.

[2] A. Ahmed, Y. Low, M. Aly, V. Josifovski, and A. J. Smola. Scalable inference of dynamic user interests for behavioural targeting. In *Knowledge Discovery and Data Mining*, 2011.

[3] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK Users' Guide*. SIAM, Philadelphia, second edition, 1995.

[4] Apache Foundation. Mahout project, 2012. http://mahout.apache.org.

[5] R. Berinde, G. Cormode, P. Indyk, and M.J. Strauss. Space-optimal heavy hitters with strong error bounds. In J. Paredaens and J. Su, editors, *Proceedings of the Twenty-Eigth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS*, pages 157–166. ACM, 2009.

[6] C. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006.

[7] D. Blei, A. Ng, and M. Jordan. Latent Dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, January 2003.

[8] J. Byers, J. Considine, and M. Mitzenmacher. Simple load balancing for distributed hash tables. In *Peer-to-peer systems II*, pages 80–87. Springer, 2003.

[9] K. Canini. Sibyl: A system for large scale supervised machine learning. *Technical Talk*, 2012.

[10] B.-G. Chun, T. Condie, C. Curino, C. Douglas, S. Matusevych, B. Myers, S. Narayanamurthy, R. Ramakrishnan, S. Rao, J. Rosen, R. Sears, and M. Weimer. Reef: Retainable evaluator execution framework. *Proceedings of the VLDB Endowment*, 6(12):1370–1373, 2013.

[11] G. Cormode and S. Muthukrishnan. Summarizing and mining skewed data streams. In *SDM*, 2005.

[12] W. Dai, J. Wei, X. Zheng, J. K. Kim, S. Lee, J. Yin, Q. Ho, and E. P. Xing. Petuum: A framework for iterative-convergent distributed ml. *arXiv preprint arXiv:1312.7651*, 2013.

[13] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, Q. Le, M. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Ng. Large scale distributed deep networks. In *Neural Information Processing Systems*, 2012.

[14] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *CACM*, 51(1):107–113, 2008.

[15] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In T. C. Bressoud and M. F. Kaashoek, editors, *Symposium on Operating Systems Principles*, pages 205–220. ACM, 2007.

[16] J. J. Dongarra, J. Du Croz, S. Hammarling, and R. J. Hanson. An extended set of fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14:18–32, 1988.

[17] The Apache Software Foundation. Apache hadoop nextgen mapreduce (yarn). http://hadoop.apache.org/.

[18] The Apache Software Foundation. Apache hadoop, 2009. http://hadoop.apache.org/core/.

[19] F. Girosi, M. Jones, and T. Poggio. Priors, stabilizers and basis functions: From regularization to radial, tensor and additive splines. A.I. Memo 1430, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1993.

[20] T.L. Griffiths and M. Steyvers. Finding scientific topics. *Proceedings of the National Academy of Sciences*, 101:5228–5235, 2004.

[21] S. H. Gunderson. Snappy: A fast compressor/decompressor. https://code.google.com/p/snappy/.

[22] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*. Springer, New York, 2 edition, 2009.

[23] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, pages 22–22, 2011.

[24] Q. Ho, J. Cipar, H. Cui, S. Lee, J. Kim, P. Gibbons, G. Gibson, G. Ganger, and E. Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *NIPS*, 2013.

[25] M. Hoffman, D. M. Blei, C. Wang, and J. Paisley. Stochastic variational inference. In *International Conference on Machine Learning*, 2012.

[26] W. Karush. Minima of functions of several variables with inequalities as side constraints. Master's thesis, Dept. of Mathematics, Univ. of Chicago, 1939.

[27] L. Kim. How many ads does Google serve in a day?, 2012. http://goo.gl/oIidXO.

[28] D. Koller and N. Friedman. *Probabilistic Graphical Models: Principles and Techniques*. MIT Press, 2009.

[29] T. Kraska, A. Talwalkar, J. C. Duchi, R. Griffith, M. J. Franklin, and M. I. Jordan. Mlbase: A distributed machine-learning system. In *CIDR*, 2013.

[30] L. Lamport. Paxos made simple. *ACM Sigact News*, 32(4):18–25, 2001.

[31] M. Li, D. G. Andersen, and A. J. Smola. Distributed delayed proximal gradient methods. In *NIPS Workshop on Optimization for Machine Learning*, 2013.

[32] M. Li, D. G. Andersen, and A. J. Smola. Communication Efficient Distributed Machine Learning with the Parameter Server. In *Neural Information Processing Systems*, 2014.

[33] M. Li, L. Zhou, Z. Yang, A. Li, F. Xia, D.G. Andersen, and A. J. Smola. Parameter server for distributed machine learning. In *Big Learning NIPS Workshop*, 2013.

[34] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed Graphlab: A framework for machine learning and data mining in the cloud. In *PVLDB*, 2012.

[35] H. B. McMahan, G. Holt, D. Sculley, M. Young, D. Ebner, J. Grady, L. Nie, T. Phillips, E. Davydov, and D. Golovin. Ad click prediction: a view from the trenches. In *KDD*, 2013.

[36] K. P. Murphy. *Machine learning: a probabilistic perspective*. MIT Press, 2012.

[37] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013.

[38] A. Phanishayee, D. G. Andersen, H. Pucha, A. Povzner, and W. Belluomini. Flex-KV: Enabling high-performance and flexible KV systems. In *Proceedings of the 2012 workshop on Management of big data systems*, pages 19–24. ACM, 2012.

[39] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In R. H. Arpaci-Dusseau and B. Chen, editors, *Operating Systems Design and Implementation, OSDI*, pages 293–306. USENIX Association, 2010.

[40] PRObE Project. Parallel Reconfigurable Observational Environment. https://www.nmc-probe.org/wiki/Machines:Susitna,

[41] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Heidelberg, Germany, November 2001.

[42] B. Schölkopf and A. J. Smola. *Learning with Kernels*. MIT Press, Cambridge, MA, 2002.

[43] A. J. Smola and S. Narayanamurthy. An architecture for parallel topic models. In *Very Large Databases (VLDB)*, 2010.

[44] E. Sparks, A. Talwalkar, V. Smith, J. Kottalam, X. Pan, J. Gonzalez, M. J. Franklin, M. I. Jordan, and T. Kraska. Mli: An api for distributed machine learning. 2013.

[45] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[46] C.H. Teo, Q. Le, A. J. Smola, and S. V. N. Vishwanathan. A scalable modular convex solver for regularized risk minimization. In *Proc. ACM Conf. Knowledge Discovery and Data Mining (KDD)*. ACM, 2007.

[47] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, volume 4, pages 91–104, 2004.

[48] V. Vapnik. *The Nature of Statistical Learning Theory*. Springer, New York, 1995.

[49] R.C. Whaley, A. Petitet, and J.J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

[50] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. M. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Fast and interactive analytics over Hadoop data with Spark. *USENIX ;login:*, 37(4):45–51, August 2012.



[32] M. Li, D. G. Andersen, and A. J. Smola. Communication Efficient Distributed Machine Learning with the Parameter Server. In *Neural Information Processing Systems*, 2014.

[33] M. Li, L. Zhou, Z. Yang, A. Li, F. Xia, D.G. Andersen, and A. J. Smola. Parameter server for distributed machine learning. In *Big Learning NIPS Workshop*, 2013.

[34] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein. Distributed Graphlab: A framework for machine learning and data mining in the cloud. In *PVLDB*, 2012.

[35] H. B. McMahan, G. Holt, D. Sculley, M. Young, D. Ebner, J. Grady, L. Nie, T. Phillips, E. Davydov, and D. Golovin. Ad click prediction: a view from the trenches. In *KDD*, 2013.

[36] K. P. Murphy. *Machine learning: a probabilistic perspective*. MIT Press, 2012.

[37] D. G. Murray, F. McSherry, R. Isaacs, M. Isard, P. Barham, and M. Abadi. Naiad: a timely dataflow system. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 439–455. ACM, 2013.

[38] A. Phanishayee, D. G. Andersen, H. Pucha, A. Povzner, and W. Belluomini. Flex-KV: Enabling high-performance and flexible KV systems. In *Proceedings of the 2012 workshop on Management of big data systems*, pages 19–24. ACM, 2012.

[39] R. Power and J. Li. Piccolo: Building fast, distributed programs with partitioned tables. In R. H. Arpaci-Dusseau and B. Chen, editors, *Operating Systems Design and Implementation, OSDI*, pages 293–306. USENIX Association, 2010.

[40] PRObE Project. Parallel Reconfigurable Observational Environment. https://www.nmc-probe.org/wiki/Machines:Susitna,

[41] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Heidelberg, Germany, November 2001.

[42] B. Schölkopf and A. J. Smola. *Learning with Kernels*. MIT Press, Cambridge, MA, 2002.

[43] A. J. Smola and S. Narayanamurthy. An architecture for parallel topic models. In *Very Large Databases (VLDB)*, 2010.

[44] E. Sparks, A. Talwalkar, V. Smith, J. Kottalam, X. Pan, J. Gonzalez, M. J. Franklin, M. I. Jordan, and T. Kraska. Mli: An api for distributed machine learning. 2013.

[45] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.

[46] C.H. Teo, Q. Le, A. J. Smola, and S. V. N. Vishwanathan. A scalable modular convex solver for regularized risk minimization. In *Proc. ACM Conf. Knowledge Discovery and Data Mining (KDD)*. ACM, 2007.

[47] R. van Renesse and F. B. Schneider. Chain replication for supporting high throughput and availability. In *OSDI*, volume 4, pages 91–104, 2004.

[48] V. Vapnik. *The Nature of Statistical Learning Theory*. Springer, New York, 1995.

[49] R.C. Whaley, A. Petitet, and J.J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.

[50] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. M. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Fast and interactive analytics over Hadoop data with Spark. *USENIX ;login:*, 37(4):45–51, August 2012.