LDA*: A Robust and Large-scale Topic Modeling System

[Innovative Systems and Applications]

Lele Yu[†,§] Ce Zhang[‡] Yingxia Shao[†] Bin Cui[†]
[†]School of EECS, Peking University
[‡]Department of Computer Science, ETH Zürich
[§]Tencent Inc.
[†]{leleyu, simon0227, bin.cui}@pku.edu.cn [‡]ce.zhang@inf.ethz.ch [§]leleyu@tencent.com

ABSTRACT

We present LDA*, a system that has been deployed in one of the largest Internet companies to fulfil their requirements of *"topic modeling as an internal service"*—relying on thousands of machines, engineers in different sectors submit their data, some are as large as 1.8TB, to LDA* and get results back in hours. LDA* is motivated by the observation that *none of the existing topic modeling systems is robust enough*—Each of these existing systems is designed for a specific point in the tradeoff space that can be suboptimal, sometimes by up to $10\times$, across workloads.

Our first contribution is a systematic study of all recently proposed samplers: AliasLDA, F+LDA, LightLDA, and WarpLDA. We discovered a novel system tradeoff among these samplers. Each sampler has different sampling complexity and performs differently, sometimes by $5\times$, on documents with different lengths. Based on this tradeoff, we further developed a hybrid sampler that uses different samplers for different types of documents. This hybrid approach works across a wide range of workloads and outperforms the fastest sampler by up to $2\times$. We then focused on distributed environments in which thousands of workers, each with different performance (due to virtualization and resource sharing), coordinate to train a topic model. Our second contribution is an *asymmetric* parameter server architecture that pushes some computation to the parameter server side. This architecture is motivated by the skew of the word frequency distribution and a novel tradeoff we discovered between communication and computation. With this architecture, we outperform the traditional, symmetric architecture by up to $2\times$.

With these two contributions, together with a carefully engineered implementation, our system is able to outperform existing systems by up to $10\times$ and has already been running to provide topic modeling services for more than six months.

1. INTRODUCTION

Bayesian inference and learning, a topic that has emerged since 1980s [17], has become relatively mature in recent years. The last decade has witnessed a wide array of applications of such technique to applications such as information extraction [28], image understanding [9], probabilistic databases [20]. In industry settings, one implication of this level of maturity is that there are many developers and engineers in the same company who need to run Bayesian inference and learning on daily basis. In this paper, we are motivated by one question that our industry partner asked: *Can you build a "Bayesian Inference as a Service"-like infrastructure to support all of our developers to run their inference and learning task, with thousands of machines?*

We find that, even if we scope ourselves to a specific type of tasks such as running topic modeling with Latent Dirichlet Allocation (LDA), this is a challenging question. Despite of the abundance of existing systems, including LightLDA [27], Petuum [8], and YahooLDA [19], none of them is designed to deal with the *diversity* and *skew* of workloads that we see from our industry partner. In this paper, we ask *How to design a single system to support a diverse set of workloads for distributed topic modeling with thousands of machines in a real-world production environment?*

Painpoint & Challenge 1. One challenge in building such a system roots from the diversity of workloads. In our use case, users need to run topic modeling on documents of very different types such as web corpus, user behaviours logs, and social network posts. Therefore, the length of documents varies—the average length of the NIPS dataset is about 1,000 tokens while it is only 90 for the PubMED dataset. This diversity of input corpus causes different systems to slow down on different datasets we were trying to support. Worse, there are no studies of this tradeoff available so far.

Painpoint & Challenge 2. One challenge in building a massively scalable system for topic modeling is caused by (1) the skewness of data, and (2) the symmetric architecture of traditional parameter servers. The word frequency in most natural language corpus follows a power law [14] distribution in which some words are orders of magnitude more frequent than others. In state-of-the-art distributed topic modeling systems, the communication cost caused by unpopular words are significantly higher than the others. However, most of existing systems are not aware of such skewness. When these systems are deployed on thousands of machines with non-InfiniBand networks, we can often observe significant performance degradation.

Motivated by these two challenges we faced while trying to deploy existing systems, we decide to build LDA* with a design that not only integrates the recent exciting advancement of distributed topic modeling [19, 8, 27], but also keeps in mind the diversity in datasets and infrastructure we saw from our industry partners. Our approach is to systematically study the system tradeoff caused by the diversity and skewness of the workloads, and design novel algorithms and system architectures accordingly. As a result, LDA* can be up to $10\times$ faster than existing systems.

Overview of Technical Contributions. We describe our technical contributions. We first present a short premier of Gibbs sampling for topic modeling. Then we present a hybrid sampler, motivated by our systematic study of recently proposed samplers, on a single machine; and then an asymmetric parameter server infrastructure to deal with the skewness of the data.

Gibbs Sampling: A Premier. Gibbs sampling is the *de facto* algorithm to solve distributed topic modeling that has been used intensively by previous work [5, 29, 25, 11, 26, 27, 2, 8, 19]. We define a corpus $\mathcal{C} = \{D_1, ..., D_n\}$ as a set of documents, and each document D_i contains a set of tokens $\{t_{ij}\}$, and let $\mathcal{T}$ be the set of all tokens. Let $\mathcal{V}$ be the vocabulary, a set of distinct words that a token can take value from, and K be the number of topics, which is usually a hyperparameter to the system. The goal of topic modeling is to assign for each token t_{ij} a topic distribution—one probability value for each of the K topics. The data structures involved in topic modeling can be represented with two matrixes: (1) $\mathbf{C_w} : \mathcal{V} \mapsto \mathbb{N}_+^K$ assigns each word to a topic distribution, and (2) $\mathbf{C_d} : \mathcal{C} \mapsto \mathbb{N}_+^K$ assigns each document to a topic distribution. In distributed setting, the standard parameter server architecture usually partitions the $\mathbf{C_d}$ matrix among workers while the $\mathbf{C_w}$ matrix is stored among parameter servers. Inside each iteration, each worker pulls the $\mathbf{C_w}$ matrix from parameter servers and pushes the updates of $\mathbf{C_w}$ back after the sampling operations.

Contribution 1: Single Machine & Sampler Selection. We first focus on a single machine and study the question that *Out of the four recently proposed Gibbs samplers, which one should we select given a corpus?* As we will see, a suboptimal selection of a sampler can be 5× slower than the optimal strategy.

(An Anatomy of Existing Samplers) We systematically study four samplers AliasLDA, F+LDA, LightLDA, and WarpLDA [2, 27, 11, 26]:

Sparsity-Aware (SA) Samplers. AliasLDA, and F+LDA exploit the sparse structure of $\mathbf{C_w}$ and $\mathbf{C_d}$ to reduce the sampling complexity—$O(K_d)$ for AliasLDA and F+LDA, where $K_d = |C_d(d)|$ is the number of topics ever picked by at least one token in a document. When the length of documents is skewed, the performance is dominated by the longest document.

Metropolis-Hastings (MH) Samplers. LightLDA and WarpLDA use Metropolis-Hastings to achieve an $O(1)$ sampling complexity. The time complexity per token of an MH sampler is orthogonal to the document length or the number of topic. However, MH samplers usually require *more than one samples* for each token.

(Summary of the Tradeoff) We discover a tradeoff between SA samplers and MH samplers. The distribution of document lengths is the key—When the datasets containing many long documents, SA samplers can be 2× slower than MH samplers; The performance gap can become more significant when we increase the number of topics. On the other hand, MH samplers need more iterations to converge to a comparable solution. For dataset with many short documents, MH samplers can be 1.5-5× slower than SA samplers. Similar phenomenon holds for the number of topics K.

(Hybrid Sampler) The study of the tradeoff space raises a natural question that *Can we design a hybrid sampler that outperforms both SA and MH samplers?* The answer is yes. We propose a simple, but novel hybrid sampler based on F+LDA and WarpLDA, and design a novel index structure to support the access pattern and parallelization strategies of both F+LDA and WarpLDA. The empirical results in Section 6 show that the hybrid sampler can consistently outperform all others over all of our datasets by up to 2×.

Contribution 2: Distributed Setting & Asymmetric Parameter Server. We observe that, when deploying existing systems to thousands of machines, each of which is shared and competing with other online applications, the performance of these systems degrades significantly, especially when the network is not as fast as InfiniBand. Our second contribution consists of three parts.

(Asymmetric Parameter Server) In traditional parameter server architecture, the system is *symmetric* in the sense that all workers conduct sampling and the parameter server is just treated as a distributed shared memory storing $\mathbf{C_w}$. In our architecture, both workers and servers have the ability to perform computation, here, the sampling operation. By pushing computation to the parameter server, we take advantage of the skewness of words on each partition to reduce communication cost (because they are sampled on the parameter server directly), which has long been ignored by existing systems. In practice, when running topic modeling on a 1.8 TB dataset from our industry partner, our asymmetric architecture reduces communication by 55% when scaling with 1000 workers.

(Skewness-aware Partition) We further notice that the skewness in natural language corpus causes unbalanced partitioning results. Therefore, we propose a skew-aware partitioning strategy for the parameter servers to balance the communication, which achieves 50% improvement on performance.

(Fine-grained Synchronization) We also study different synchronization strategies. Existing systems, such as Petuum and LightLDA, require global barrier for updating the $\mathbf{C_w}$ matrix. In our scenario, this introduces heavy overhead especially in the presence of resource sharing. In our architecture, we introduce a more finer-grained synchronization strategy that partitions $\mathbf{C_w}$ into multiple *synchronization units* and only synchronize one unit of $\mathbf{C_w}$ at a time. This allows us to hide the synchronization latency along with computation of other units.

All these techniques together lead to 2-5× speed up over existing systems. This holds robustly across a range of different infrastructures with different network speed.

Contribution 3: System Implementation and Empirical Evaluation. We develop `LDA*` based on the study of system tradeoffs for both a single machine and multiple machines. Along with the previous two technical contributions, we also describe engineering considerations and design decisions involved in building such a system. Our system has been running steadily over a real production cluster of our industry partner for six months, and has been used to support users from a diverse set of internal sectors on a daily basis.

We evaluate our system and compare it with other systems on a range of applications. On a dataset provided by our industry partner, `LDA*` can train topic model over a 1.8 TB dataset with 308 billion tokens in about 4 hours by using 1,000 workers which are allocated with only 8 GB memory. To the best of our knowledge, no existing systems can achieve such performance for terabytes of data. We also conduct experiments with a set of standard benchmark datasets—on these datasets, `LDA*` can be up to 10× faster than the existing systems.

Overview. The rest of this paper is organized as follows. We present our study of existing samplers in Section 3.1 and describe the hybrid sampler in Section 3.2. We further describe our distributed architecture and system implementation in Section 4 and Section 5 respectively. We show the experimental results in Section 6 and discuss related work in Section 7.

2. PRELIMINARIES

We present preliminary materials in this section. We start by describing topic modeling with LDA and how to solve it with Gibbs sampling. We then present background on the basics of Metroplis-Hastings, a general case of Gibbs sampling that many state-of-the-art algorithms used to further optimize their samplers.

2.1 Latent Dirichlet Allocation

LDA is a generative probabilistic model that runs over a collection of documents (or other discrete sets). A *corpus* $\mathcal{C}$ is a collection of M documents $\{D_1, \cdots, D_M\}$ and each *document* D_i is a sequence of N_i tokens denoted by $D_i = (t_1, t_2, \cdots, t_{N_i})$, where each *token* t_n is a *word*. Each *word* w is an item from a vocabulary set $\mathcal{V}$. Let K be the number of topics, LDA models each document as random mixtures over latent topics. Formally, each document d is represented as K-dim topic distribution θ_d while each topic k is a V-dim word distribution ϕ_k, which follows a Dirichlet prior β.

To generate a document d, LDA first draws a K-dimensional topic mixing distribution $\theta_d \sim Dir(\alpha)$, where α is a hyperparameter. For each token t_{dn}, it first draws a topic assignment z_{dn} from a multinomial distribution $\mathbf{Mult}(\theta_d)$, then it draws a word $w_{dn} \in \mathcal{V}$ from $\mathbf{Mult}(\phi_{z_{dn}})$.

We further denote $\mathbf{Z} = \{\mathbf{z}_d\}_{d=1}^{D}$ as the topic assignments for all tokens, where $\mathbf{z}_d = \{z_{dn}\}_{n=1}^{L_d}$, and $\mathbf{\Phi} = [\phi_1 \cdots \phi_V]$ be the topic word matrix with $V \times K$ dimensions. We use $\mathbf{\Theta} = [\theta_1 \cdots \theta_D]$ to denote the $D \times K$ document topic matrix. The inference process of LDA is to obtain the posterior distribution of latent variables $(\mathbf{\Theta}, \mathbf{\Phi}, \mathbf{Z})$ given observations $\mathcal{C}$ and hyperparameters α and β. Collapsed Gibbs Sampling (CGS) integrates out $(\mathbf{\Theta}, \mathbf{\Phi})$ through conjugacy and iteratively samples z_{dn} for tokens from the following full conditional distribution:

$$p(z_{dn} = k | t_{dn} = w, \mathbf{Z}_{\neg dn}, \mathcal{C}_{\neg dn}) \propto \frac{C_{wk}^{\neg dn} + \beta}{C_k^{\neg dn} + V\beta} (C_{dk}^{\neg dn} + \alpha) \quad (1)$$

where C_{dk} is the number of tokens that are assigned to topic k in document d; C_{wk} is the times that word w is assigned to topic k; $C_k = \sum_w C_{wk} = \sum_d C_{dk}$. The superscript or subscript $\neg dn$ represents that z_{dn} or t_{dn} is excluded from the count value or collections. Moreover, we use $\mathbf{C_w}$ to represent the $V \times K$ matrix formed by all C_{wk} and $\mathbf{C_d}$ to stand for the $D \times K$ matrix formed by all C_{dk}. $\mathbf{C_w}[w,]$ and $\mathbf{C_d}[d,]$ represent the particular rows indexed by w and d. Figure 1 gives a simple example for a corpus with three documents.

Core Operation. During the inference phase of LDA, CGS iteratively assigns topics for tokens in $\mathcal{C}$. For one token, it calculates out all probabilities for K topics according to Eq. 1 and then randomly picks a new one. We call this process the *core operation*. This induces an $O(K)$ computation complexity per token and is very inefficient for applications with massive tokens and large value of K. For more details, readers can refer [4]. After burn-in, CGS is able to generate samples that follow the posterior distribution $p(\mathbf{Z}|\mathcal{C}, \alpha, \beta)$. We can use these samples to estimate the distribution of $\mathbf{Z}$, $\mathbf{\Theta}$ and $\mathbf{\Phi}$, which allow us to understand the semantic information of documents and words.

Epoch. An *epoch* is one complete pass of dataset in which all tokens are sampled by the core operation once. The inference phase contains tens, often hundreds, of epochs.

The process of Gibbs sampling is to run many, often tens or hundreds, of epochs over the dataset. Each epoch executes the core operation to sample its topic assignment for each token and estimates the distribution of $\mathbf{Z}$, $\mathbf{\Theta}$ and $\mathbf{\Phi}$ using the generated samples.

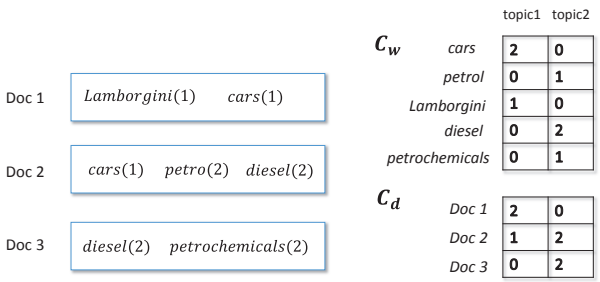


Figure 1: An example for Corpus. The numbers in the parentheses are the current topic of each token. $\mathbf{C_w}$ and $\mathbf{C_d}$ contain counts of tokens that belong to a specific topic.

Algorithm 1: Metropolis-Hastings algorithm

Input : $p(x)$, $q(x)$, number of steps M
Initialize $x^{(0)} \sim q(x)$
for $i \leftarrow 1$ **to** M **do**
  Propose $x^{cand} \sim q(x^{(i)}|x^{(i-1)})$
  Acceptance rate $\pi = min\{1, \frac{p(x^{cand})q(x^{(i-1)}|x^{cand})}{p(x^{(i-1)})q(x^{cand}|x^{(i-1)})}\}$
  if *Uniform(0,1)* $\leq \pi$ **then** $x^{(i)} = x^{cand}$
  else $x^{(i)} = x^{(i-1)}$

2.2 Metropolis-Hastings

Directly sampling from probability distribution of Eq.1 is expensive. Here we describe an efficient method to reduce the sampling complexity, which is called *Metropolis-Hastings* (MH) algorithm.

Let $p(x)$ be the target distribution we want to draw samples from. MH method constructs a Markov chain with an easy-to-sample proposal distribution $q(x)$. Starting with an arbitrary state $x^{(0)}$, MH repeatedly generates samples from the proposal distribution $x^{(i)} \sim q(x^{(i)}|x^{(i-1)})$ at each step i, and updates the current state with the new sample with an *acceptance rate*

$$\pi = min\left\{1, \frac{p(x^{(i)})q(x^{(i-1)}|x^{(i)})}{p(x^{(i-1)})q(x^{(i)}|x^{(i-1)})}\right\}.$$

Algorithm 1 presents the detail of Metropolis-Hastings. We call the hyperparameter M an *MH steps*. Under certain technical condition, $q(x^{(i)})$ converges to $p(x)$ as $i \to \infty$, regardless of $x^{(0)}$ [10]. Gibbs sampling is a special case of Metropolis-Hastings.

3. SAMPLER SELECTION

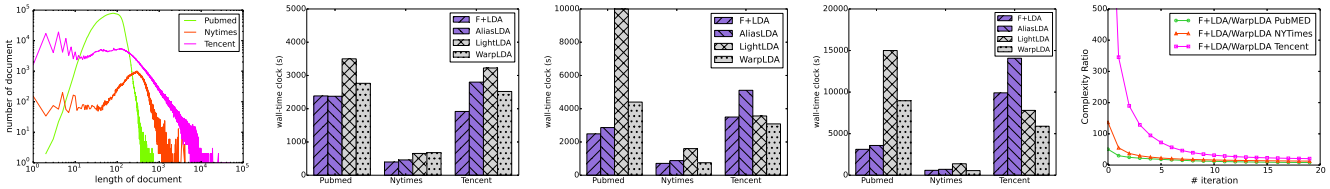
We study the system tradeoff of different samplers and propose a novel hybrid approach that automatically decides which sampler to use for each document. We start by presenting a taxonomy of four existing samplers, all of which were published recently.

3.1 Anatomy of Existing Samplers

Existing samplers can be classified into two categories according to the optimizations they use to reduce the sampling complexity: (1) **SA samplers**, including AliasLDA and F+LDA, exploit the sparse structure of $\mathbf{C_d}$; and (2) **MH samplers**, including LightLDA and WarpLDA, use Metropolis-Hastings to scale to a large number of topics. We study their tradeoff with the following experiment setup.

Settings. We implement all samplers under the same code base. For fair comparison, we optimize all these techniques and they are faster than all their original open-source implementations.[1]. All

[1] https://github.com/Microsoft/LightLDA,http://bigdata.ices.utexas.edu/software/nomad/,https://github.com/thu-ml/warplda



(a) doc length distribution (b) K = 100 (c) K = 1k (d) K = 16k (e) complexity over datasets

Figure 2: Effects of document length and K on the performance of samplers

experiment results that we report are on a single machine and all samplers are parallelized with 16 threads.

Metrics. We measure the performance by the wall-clock time a sampler requires to converge to a given log likelihood value. We vary the number of topics and use datasets that have different characteristics to measure each sampler's performance.

Datasets. We use three different datasets to illustrate the tradeoff. One key characteristic turns out to be the length of a document, and the document length distribution is showed in Figure 2(a). We see that these datasets have different length distribution—the PubMED dataset consists of many short documents whose length is less than 200, while the NYTimes dataset contains both short documents and long documents whose length can vary from 100 to 2000. For the Tencent dataset (the one from our industry partner), most of the tokens are from long documents with length larger than 1000—about 24% of documents have a length over 1000, which contains more than 76% of all tokens.

3.1.1 Sparse-Aware (SA) Samplers

AliasLDA and F+LDA decompose the probability for each token (Eq. 1) into two parts: $C_{dk}\frac{C_{wk}+\beta}{C_k+V\beta}$ and $\alpha\frac{C_{wk}+\beta}{C_k+V\beta}$. When C_{dk} ($\mathbf{C_d}$) is sparse, the sampler can skip those with zero C_{dk}, thus lowering the complexity. The difference between these two algorithms is the different data structures they use to perform sampling—AliasLDA uses the Alias table [22], while F+LDA uses the F+ tree.

Results. Figures 2(b), 2(c), and 2(d) show the results for different number of topics K. We can see that F+LDA is consistently faster than AliasLDA over all datasets and various values of K. The relative performances of AliasLDA and F+LDA are different across different datasets. Specifically, the gap becomes larger on the Tencent dataset, while it is much smaller on the other two datasets. This is because AliasLDA requires both traversal access and random access for $\mathbf{C_d}$ matrix, while F+LDA only requires traversal access. Since $\mathbf{C_d}$ is stored in sparse format—which aims to reduce memory cost—the latency of random access increases with more items in one document due to the increased conflicts in the hashtable.

3.1.2 Metropolis-Hastings (MH) Samplers

LightLDA and WarpLDA draw samples from two proposal distributions — $q^{doc} \propto C_{dk}+\alpha$ and $q^{word} \propto \frac{C_{wk}+\beta}{C_k+\beta}$ — alternatively, and accept their proposals based on the acceptance condition of Metropolis-Hastings. These two samplers are different in their access order for the in-memory data structures—WarpLDA separates the access of tokens into two passes, where each pass only accesses $\mathbf{C_w}$ or $\mathbf{C_d}$, respectively, in order to reduce random access.

Results. Figures 2(b), 2(c), 2(d) show the results of MH Samplers. We see that WarpLDA is consistently faster than LightLDA. This is because WarpLDA effectively eliminates the amount of random accesses. The performance gap is influenced by the size of the vocabulary $|\mathcal{V}|$ — the more words the corpus has, the more random accesses will be incurred by $\mathbf{C_w}$. For the PubMED dataset, which has the largest vocabulary, the performance gap between WarpLDA and LightLDA is largest; on the other hand, since the Tencent dataset only contains 88,916 distinct words in its vocabulary, this performance gap is relatively small.

3.1.3 Tradeoff: SA vs. MH Samplers

We now describe the system tradeoff between SA and MH samplers. Because in our experiments F+LDA and WarpLDA always dominate others, we focus on the tradeoff between them. We first describe the tradeoff we observed, and then analyze it.

Summary of the Tradeoff

1. **Length of Document L_d.** The length of documents turns out to be an important axis in the tradeoff. On datasets with many short documents, like PubMED, F+LDA is faster than WarpLDA—by up to 2.8× when $K = 16k$; (2) On datasets with many long documents, like Tencent, WarpLDA is faster than F+LDA— by up to 1.7× when $K = 16k$.

2. **Number of topics K.** The number of topics also plays a major role in the tradeoff. When the number of topics increases—e.g., from $1k$ to $16k$ on Tencent— WarpLDA becomes faster compared with F+LDA; Similarly, when the number of topics increases on PubMED, F+LDA becomes faster compared with WarpLDA.

Analysis. The above tradeoff can be explained with the following analytical model. To generate one sample in WarpLDA, the Metropolis-Hastings sampler needs to decide whether to accept a sample or not. Let π be the acceptance rate, WarpLDA requires, in expectation, $\frac{1}{\pi}$ samples for one sample to be accepted. On the other hand, F+LDA does not have this overhead. However, to generate each sample, the computational complexity of F+LDA is $O(K_d)$, where K_d, the number of non-zero elements of C_d is bound by the number of topics K and the length of a document L_d. On the other hand, WarpLDA incurs $O(1)$ complexity to propose each sample.

We can now see the tradeoff—when K and L_d are small, WarpLDA is slower because of its $\frac{1}{\pi}$ overhead; otherwise, F+LDA is slower due to its overhead in generating each sample.

The result we see in Figures 2(b), 2(c), 2(d) is the result of an *aggregation* of the above analysis across the whole corpus. Figure 2(a) illustrates the distribution of the length of documents. For the PubMED dataset, which consists of many documents with small L_d, F+LDA is faster than WarpLDA. For the Tencent dataset, which consists of documents with large L_d, WarpLDA is faster than F+LDA. For smaller K, F+LDA is faster than WarpLDA on all datasets.

Figure 2(e) further provides a quantitative illustration of the tradeoff. We define a *complexity ratio* $\lambda = K_d\pi$ as the ratio between the complexity of F+LDA and WarpLDA. As expected, on datasets where F+LDA is faster, λ is smaller. This is consistent with the empirical result illustrated in Figure 2(e).

3.2 Hybrid Sampler

The above tradeoff raises a natural question— *Can we build a hybrid sampler that marries both SA and MH sampler?* Intuitively, this is trivial—just run "short" documents with F+LDA and

"long" documents with WarpLDA. In practice, however, there are two technical questions: (1) how to decide which document is *long enough* to "qualify" for WarpLDA; and (2) how to *balance* an MH sampler and an SA sampler, which have different convergence speeds. For the first question, we developed a very simple rule-of-thumb based on the study of our tradeoff. The second question is more challenging and ties to an open question in the mixing theory of two Markov chains. Inspired by classic statistical theory on a simpler underlying model, we develop a simple heuristics that works well across all of our datasets.

3.2.1 Sampler Selection

The first step is to design a rule-of-thumb to choose between two samplers. We focus on the combination of F+LDA and WarpLDA, as these two samplers dominate consistently faster than AliasLDA and LightLDA in our tradeoff study.

Sampling Complexity. For F+LDA to generate one sample, it needs to traverse the non-zero items in C_{dk}, whose size is bounded by K and L_d. Thus, the sampling complexity of F+LDA is

$$C_{f+} = O(min(L_d, K))$$

On the other hand, since Metropolis-Hastings method requires mix time for each sample, the complexity of WarpLDA is

$$C_{warp} = O(n)$$

where n is the steps to achieve mix.

The technical challenge is how to estimate n, whose value not only depends on the input datasets, but also changes across iterations. In theory, estimating the *mixing time* of the underlying MCMC chain for general factor graphs is a problem that has been around for decades. With this in mind, we resort to a simple heuristics that is motivated by the empirical observation from our study.

Heuristics 1. Given a document d with length L_d and topic number K, if $K \leq S$ or $L_d \leq S$ choosing F+LDA; otherwise, choosing WarpLDA. S is a threshold value that depends on the implementation and properties of the dataset.

3.2.2 Balancing Two Chains

One surprising observation is that the above heuristics itself is not enough for a hybrid sampler that outperforms both F+LDA and WarpLDA. The fundamental reason is that two Markov chains underlying F+LDA and WarpLDA *do not converge with the same speed*. Specifically, all samples from F+LDA will be accepted, however this is not the case for WarpLDA. Intuitively, this means that a naive hybrid sampler would generate more samples for tokens belonging to F+LDA per epoch than WarpLDA.

Theoretical Understanding. The above observation raises a fundamental question: *What is the relationship between the mixing time of a Gibbs sampler (F+LDA) and a Metropolis-Hastings (WarpLDA) sampler for the same underlying distribution?* If we *magically* know the ratio between their mixing time, we could just use this number to balance these two chains.

Unfortunately, a general treatment of this question has existed for decades and is still under intensive study by the theoretical computer science and mathematics community. However, there are theoretical results that can be used to *inspire* our practical heuristics.

We know that the mixing time of these two chains is not too different, at least for Ising model:

LEMMA 3.1. *[10](Levin, Peres, & Wilmer 2008, Example 13.18) For a graph with vertex set V, let π be the Ising probability measure. Let γ be the spectral gap of the Gibbs chain and $\tilde{\gamma}$ be the spectral gap of the Metropolis chain using the base chain, we have*

$$\gamma \leq \tilde{\gamma} \leq 2\gamma$$

where γ is related to the mixing time by the following lemma

LEMMA 3.2. *[10](Levin, Peres, & Wilmer 2008, Theorem 12.3) Let P be the transition matrix of a reversable, irreducible Markov chain with state space Ω, and π be the underlying probability measure. Let $\pi_{min} = \min_{x\in\Omega} \pi(x)$ and γ the absolute spectral gap (equals to the spectral gap when the chain is "lazy"), we have the mixing time*

$$t_{min}(\epsilon) \leq \log(\frac{1}{\epsilon\pi_{min}})\frac{1}{\gamma}$$

The above two lemmas inspired the design in two ways. First, we know, at least intuitively, whatever scheme we use to balance these two chains, that scheme should not be "too extreme" (these two chains are off by a constant factor anyway). Second, the mixing time is likely to be linear (or inverse linear) to the constant that we will use to balance these two chains. Inspired by these, our simple heuristics is as follows:

Heuristics 2. For each epoch, the MH steps for the WarpLDA is set to $\lceil \frac{1}{\pi} \rceil$, where π is the acceptance rate for the last epoch.

The intuition behind this heuristics is simple—just use the empirical acceptance rate $1/\pi$ as the proxy for the ratio of spectral gap $\frac{\tilde{\gamma}}{\gamma}$ (In extreme cases where all MH samples are accepted ($\pi = 1$), MH becomes Gibbs, and therefore $\frac{\tilde{\gamma}}{\gamma} = \frac{1}{\pi}$ in this extreme case.)

3.2.3 Implementation

We now present the implementation of our hybrid sampler. We design different data structures for both F+LDA and WarpLDA since they have different access patterns. Our implementation for F+LDA is actually faster than the original F+LDA paper—in order to obtain $O(min(K, L_d))$ complexity for F+LDA, the structure for short documents requires careful design, since the cost for traversing a hashmap is proportional to the size of it, which can be larger than K or L_d. Therefore, we build an extra index to guarantee achieving the theoretical complexity for F+LDA.

Structure for Short documents. Figure 3 shows the structure for short documents, which is constructed by the CSC format for accessing tokens and a set of hashmaps for accessing C_{dk}.

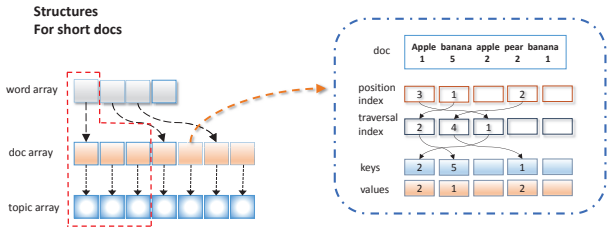


Figure 3: data structures for short docs.

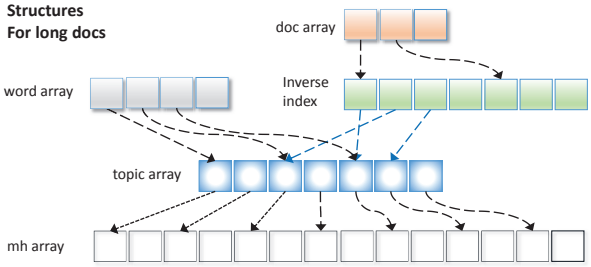


Figure 4: data structures for long docs.

Algorithm 2: Parallel Execution for Hybrid Sampler

Function `Main()`
- $ShortSet, LongSet \leftarrow$ split through Heuristics 1
- $ShortDS, LongDS \leftarrow$ build($ShortSet$), build($LongSet$)
- $mh_a = 2; mh_g = 2$
- **for** $iteration \leftarrow 1$ **to** I **do**
  - // Sampling short docs using F+LDA
  - **for** $w \leftarrow 0$ **to** V **do**
    - **for** *token* $t_{d,w}$ *belongs to* w *from* $ShortDS$ **do**
      - lock(d)
      - Perform sampling operation
      - unlock(d)
  - // Sampling long docs using WarpLDA
  - $\pi \leftarrow$ Word pass on $LongDS$
  - $mh_a = mh_g; mh_g = \lceil \frac{1}{\pi} \rceil$
  - $\pi \leftarrow$ Doc pass on $LongDS$
  - $mh_a = mh_g; mh_g = \lceil \frac{1}{\pi} \rceil$

The index is built to accelerate the traversal of non-zero items for each row of $\mathbf{C_d}$, which can guarantee the complexity of F+LDA is $O(min(K, L_d))$ rather than the size of a hashmap. Since C_{dk} can be updated after each sampling operation, we allocate one more "position array", which records the position of each key in the index array, in order to obtain an $O(1)$ maintenance complexity for each update operation. To reduce the memory introduced by the extra index, we use different data types to store these two arrays for different documents. Specifically, if $L_d \leq 256$, we use the Byte type to store them. Otherwise, the Short type is chosen.

Structure for Long documents. For long documents, the structure is the same as WarpLDA, which is presented in Figure 4. Three arrays, including word array, doc array, and inverse index, are allocated to support both the "document pass" and the "word pass" for tokens. The topic array stores the topic assignment and the MH array maintains the MH proposals for each token. The difference is that we employ this structure to visit tokens only from long documents, while WarpLDA builds it to traverse all tokens.

Parallel Execution. Since the original F+LDA and WarpLDA use different parallelization strategies, we combine them sequentially in a coarse granularity—we first sample tokens from short documents with F+LDA, then use WarpLDA to sample tokens from long documents. Algorithm 2 shows the details.

At the start of training, we initialize two MH steps, including mh_a and mh_g, which indicate the MH steps for the proposal accepting phase and the MH steps for the proposal generating phase in WarpLDA. We choose 2 as the default value for both of them since π is always less than 1. For each iteration, we first visit short documents through F+LDA in parallel and then sample tokens from long documents through WarpLDA. We parallelize the training of F+LDA over word and lock on the access of $\mathbf{C_d}[d,]$ before the sampling operation. The lock operation aims to prevent concurrent write operations for the same row of $\mathbf{C_d}$. For long documents, the parallelization is achieved by paralleling over words in the word pass and paralleling over docs in the document pass. After each pass, we calculate the value of π for it and change $mh_a = mh_g, mh_g = \lceil \frac{1}{\pi} \rceil$.

4. DISTRIBUTED ARCHITECTURE

In this section, we describe our approach that leads to an *asymmetric* parameter server architecture. Our architecture is motivated by the pattern we observed in the production environment.

In many datasets that are related to natural language, the word frequency is skew. In the distributed setting, the frequency of words

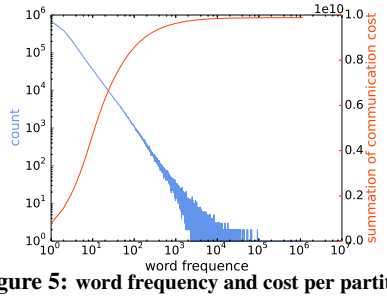


Figure 5: word frequency and cost per partition

Algorithm 3: Standard Parameter Server [24, 12, 27]

Function `WorkerIteration()`
- **for** $w \leftarrow 0$ **to** V **do**
  - $\mathbf{C_w}[w,] \leftarrow$ Pulling w
  - Sampling tokens belongs to w
  - Pushing updates of $\mathbf{C_w}[w,]$ to servers

is also skew inside each data partition. In existing parameter server systems, the communication cost is mainly induced by the *pulling* $\mathbf{C_w}$ operation for the low-frequency words. Figure 5 shows the frequency distribution of words for a randomly picked partition and the communication cost generated by words with different frequency.[2] From this figure, we see that most of communication cost is introduced by these unpopular words. Specifically, 85% of the cost is introduced by words with frequency less than 100, while only 6% of the tokens contribute to this.

4.1 Asymmetric Parameter Server

The above observation poses a fundamental challenge to the existing parameter server architecture for LDA, in which workers deal with the computation and servers deal with the distribution of parameters. As long as the architecture is designed in this form, the communication cost caused by unpopular words cannot be avoided.

This motivates us to design a new architecture— *In order to reduce the communication cost introduced by low-frequency words, we push the computation onto the servers.* This results in an *asymmetric parameter server* architecture in which servers also have the ability to do computation rather than just distributed memory storage. Pushing the computation of tokens to servers raises two technical questions: (1) how to determine which word is unpopular enough to be pushed to servers; and (2) how to design the communication protocol between the workers and servers. We first describe the execution model in our asymmetric architecture and then present how we resolve these two questions.

4.1.1 Execution Model

Figure 6 illustrates the difference between an asymmetric architecture and the standard parameter server architecture. The difference in an asymmetric architecture is that a parameter server also has the ability to conduct computation. By pushing some computation from workers to servers, we can achieve speed up over the standard symmetric architecture. Moreover, the messages communicated through the network are also different. For the standard architecture, only $\mathbf{C_w}$ and Δ_{C_w} are transferred via network since the sampling operation only happens on workers. However, for an asymmetric architecture, although both $\mathbf{C_w}$, $\mathbf{C_d}$ and Δ_{C_w} and Δ_{C_d} are transferred through network, the overall communication cost is smaller.

With an asymmetric parameter server architecture, the computation logic of each iteration becomes more complicated than a tra-

[2]The topic number K is set to 8,000 and the number of partitions is 1,000.

Algorithm 4: Asymmetric Parameter Server

Function `WorkerIteration()`
- $Pull_w, Push_w \leftarrow$ split words
- Pushing requests for $w \in Push_w$
- **for** $w \in Pull_w$ **do**
  - $\mathbf{C_w}[w,] \leftarrow$ Pulling w
  - Sampling tokens belongs to w
  - Pushing updates of $\mathbf{C_w}[w,]$ to servers
- Receiving pushing response and update $\mathbf{C_d}[d,]$

Function `ServerIteration()`
- **while** *receiving pushing request* **do**
  - Sampling tokens
  - updates $\mathbf{C_w}$
  - Pushing back topic assignments

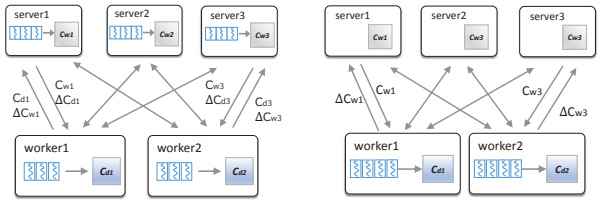


Figure 6: Architecture of Asymmetric PS and Standard PS.

ditional architecture. Algorithm 3 shows the logic of a traditional parameter server, and Algorithm 4 shows the logic of our approach.

Stage 1: Pushing $\mathbf{C_d}$ and pulling $\mathbf{C_w}$. At the start of one iteration, each worker splits the words into two parts. Then it pushes the corresponding rows of $\mathbf{C_d}$ for tokens that will be sampled on servers and pulls rows of $\mathbf{C_w}$ for tokens that will be sampled on workers.

Stage 2: Sampling. Server conducts sampling for tokens after the receiving of $\mathbf{C_d}$ and worker conducts sampling after fetching $\mathbf{C_w}$.

Stage 3: Updating. Server updates $\mathbf{C_w}$ and generates updates for $\mathbf{C_d}$, while worker updates $\mathbf{C_d}$ and generates updates for $\mathbf{C_w}$.

Stage 4: Pushing the updates of $\mathbf{C_d}$ and $\mathbf{C_w}$. The generated updates of $\mathbf{C_d}$ are pushed back to workers and the updates of $\mathbf{C_w}$ are pushed to servers from workers.

Deciding Low-Frequency Words. The challenge of determining which words are low-frequency to be sent to servers is that the sizes of $\mathbf{C_w}$ and $\mathbf{C_d}$ keep changing during the running of the algorithm. This requires a dynamically adjusting strategy to make decisions. Therefore, we maintain a "size cache" that records the exact size for each row of $\mathbf{C_w}$ on the worker side, which is updated every iteration. For word w sampled on worker, we calculate the size of $\mathbf{C_w}[w,]$ from the response message of the pulling operation. For word w sampled on the server, its size is returned from the server at stage 4.

Message Encoding. We find that we need to carefully encode the pushing requests—naively attaching one row of $\mathbf{C_d}[d,]$ for one token $t_{d,n}$ can still generate unnegligible pushing communication cost. To reduce the cost of pushing requests, we only send one pushing request for one server at each worker. The goal is to share the $\mathbf{C_d}[d,]$ for all tokens belonging to document d. Moreover, since the sampling also requires the $\mathbf{C_k}$ matrix, which is only stored on one server, we attach the value of $\mathbf{C_k}$ for one pushing request. The pushing request is encoded in one array. We organize the tokens and $\mathbf{C_d}$ document by document. For each document, we first write the value of $\mathbf{C_d}[d,]$ in the request and then write the topics assignments for tokens belonging to this document. In this way, the sampling order on the server is organized through the document order. Once we finish the sample on the document, the memory for storing $\mathbf{C_d}[d,]$ can be recycled to save memory.

Integrating with Hybrid Sampler. Our hybrid sampler can be integrated with our asymmetric parameter server architecture. There are two differences when compared to the single machine setting. First, only F+LDA is employed on the server because (1) building Alias table for each document is expensive on the server; (2) only tokens from short documents will be pushed to server because pushing tokens from long documents will increase the pushing cost. Second, we build an extra Alias table to perform the sampling operation on the word pass for WarpLDA, since the topic assignments in one worker only contain a portion of tokens belonging to one word. Directly deploying WarpLDA on each worker will generate the wrong result.

4.2 Load Balancing

Other than decreasing the communication cost through the asymmetric architecture, we also balance the load of server and worker by taking into consideration the skewness of word frequency. Otherwise, unbalanced partitioning will result in significantly more memory consumption on the workhorse machine. We describe our skew-aware partitioning strategy that tries to enforce a balanced load across machines with a simple cost model.

Partitioning of $\mathbf{C_w}$. Existing parameter server systems partition $\mathbf{C_w}$ using either range partitioning [12] or hash partitioning [6]. Range partitioning can facilitate the sequential access of Gibbs algorithm but it will generate unbalanced results, while Hash partitioning incurs random access to the data structures since fetching rows of $\mathbf{C_w}$ is out of order. Here we adopt a two-level partition algorithm that first partitions $\mathbf{C_w}$ into consecutive partitions through range partitioning and then assigns each partition to servers. To avoid generating unbalanced results, we vary the number of words in each partition at the range partitioning phase, since we can estimate the cost of each word. For a given word w, the communication cost $cost_w$ is the smaller value between $pull_w$ and $push_w$, where $pull_w$ is the cost of pulling word w from server, while $push_w$ is the cost of pushing w to server.

$$\begin{aligned} cost_w &= min(pull_w, push_w) \\ &= min(min(K, d_w) \times min(P, d_w), d_w \times L_d) \end{aligned}$$

where d_w is the degree of word w and P is the number of partitions. Thus, the number of words in each partition can be adjusted to generate partitions that incur approximately the same cost. Then we traverse the generated partitions and assign them to servers one by one. At each step, we choose the server with the minimum cost to place a new partition.

Partitioning of $\mathbf{C_d}$. The partitioning of $\mathbf{C_d}$ is determined by the partitioning of documents. Since the distribution of document length is more balanced than the frequency of word, we partition the documents through hash partitioning. The detail of the implementation is described in Section 5.2.

Synchronization. Existing systems, like YahooLDA and LightLDA, require a global barrier to synchronize the multiple copies of $\mathbf{C_w}$ on different workers, which induces extra network waiting overhead. By partitioning $\mathbf{C_w}$ into different partitions with each one containing consecutive words, we employ a fine-grained synchronization mechanism to overlap the synchronization operation and computation. First, the sampling operations are divided into multiple small units where each one is exactly one partition of $\mathbf{C_w}$. Therefore, after one worker finishes sampling one partition (both

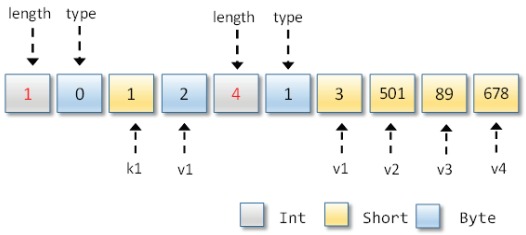


Figure 7: push and pull message encoding

on workers and servers), its updates for $\mathbf{C_w}$ are pushed to servers and synchronization is started for this partition. Hence, the synchronization of different partitions can take place at different times. Most times, when the computation of one iteration is finished, the synchronization operation can also be completed at the same time.

5. SYSTEM IMPLEMENTATION

In this section, we describe the implementation details of LDA*. We built LDA* with the architecture described in Section 4. Besides workers and servers, we employ a master node to manage the running states of them. Users submit their tasks and then LDA* is launched on Yarn. LDA* reads data stored on HDFS and dumps the model to HDFS after finishing. We use Netty and Protobuf to construct the Remote Procure Call framework for LDA*.

5.1 Encoding of Messages

To further reduce the network cost, we compress the network messages according to the distribution of words and the sparsity of $\mathbf{C_w}$ and $\mathbf{C_d}$. The sparse property enables us to use different formats to encode rows with different size. The skew distribution of words indicates that the frequency of most words is small. Since $\sum_k C_{wk} = d_w$, the value of C_{wk} can be stored with Byte type for all k if $d_w \leq 256$.

Figure 7 gives one example of our encoding method. There are two rows in this example. The first element of each row is the length value. If the length of this row equals to K, then this row is encoded by dense format. Otherwise, this row is encoded in sparse format. For each row, If the maximum value is less than 256, we encode all the elements as Byte type. The second element is one byte value, which indicates the data type of this row. For sparse format, the key is required to be stored and its range is $[0, K]$. Usually, the number of topics to be inferred varies from 5000 to 10000. Hence, we use Short type to store the key. This method of encoding is easy to construct and can significantly reduce the communication cost. This encoding format is used both for pushing $\mathbf{C_d}$ and pulling $\mathbf{C_w}$ in each iteration.

5.2 Data and Parameter Partition

LDA* supports automatically partitioning both documents and parameters. The partitioning happens at the start of training.

Partition of Documents. The goal of documents partitioning is to avoid stragglers by generating balanced results. In production clusters, the input data is stored as multiple files on HDFS (or other types of distributed file system), and each file is splitted into multiple blocks with each one stored on different nodes. Instead of incurring an extra shuffle phase, here we employ the location and meta information of each block to achieve balance. In the training process, the computation load is proportional to the number of tokens. Therefore, we can balance the computation of workers by balancing the size of each partition. So, we first collect the information of all blocks from the NameNode of HDFS, including the length of each block and the node locations. Then we can calculate the upper bound of data size for each partition. We then employ a heuristics algorithm, which assigns blocks to workers one by one. At each step, we find out the worker which shares most blocks residing on the same node of the current block. If the data size of this worker is lower than the upper bound, we assign this block to this worker. Otherwise, we assign this block to the worker with the smallest computation load.

Partition of Parameters. To implement the algorithm described in Section 4.2, we need the frequency distribution for the dataset, which can be obtained by a sequential scan of the data. In practice, we allocate another shared matrix on servers to derive the word frequency distribution. Then, we can perform the partitioning operation for $\mathbf{C_w}$. LDA* enables users to create and allocate a shared matrix during the running process. The partitioning phase will generate a list of partition keys, where each key contains the meta information of the partition. Each server will fetch the partition keys belong to it from the master and the routing information which maps the partition keys to the location of servers will be broadcasted to all workers. After that, we can start training.

5.3 Fault Tolerance

Our system provides tolerance ability for errors and machine failures. For network errors, there is a checking thread at worker which periodically monitors all network requests. If any one is timed out due to server errors or package loss, this thread will retry the request. If this request still fails after a couple of repeated attempts, worker reports the server failure to the master.

We provide mechanisms to handle failures for three main roles in LDA*, including master, server and worker. Since their computation and computation patterns are different, we design different mechanisms. Since both $\mathbf{C_w}$ and $\mathbf{C_d}$ are just count matrices which can be recomputed from the topic assignments of tokens, we only take snapshots of topic arrays during execution.

Server: The failure of a server can be detected by the interruption of heartbeat messages or the reports from workers. In case of server crash, the master reallocates a container and starts a new server. Then all workers recompute the partitions of $\mathbf{C_w}$ which belong to the crashed server and then push them to the new one.

Worker: The failure of a worker can only be detected by the timeout of heartbeat messages. Once a worker encounters any error, the master starts a new worker. The new one reloads the documents from HDFS. Furthermore, it loads the snapshot of topic arrays and rebuilds the data structures required by the hybrid sampler. In order to guarantee data consistency, we need to recompute $\mathbf{C_w}$ according to the topic arrays. Therefore this worker cleans up the value of $\mathbf{C_w}$ stored at servers through the interface of LDA*. After that, all workers recompute the values of $\mathbf{C_w}$ and push it to servers. After the rebuilding of $\mathbf{C_w}$, the computation can continue.

Master: The failure of master can be detected by the Resource Manager (RM) of Yarn. The master maintains both static and dynamic informations of a running job. The static information includes the data partitioning and parameter partitioning information while the dynamic information contains the running states of workers and servers. The static information only needs to be dumped once after the initialization of job while the dynamic information requires to be written into HDFS periodically during the running time. Once the master is crashed, the RM will kill all workers and servers then restart a new master. The new master reloads both static and dynamic informations from HDFS and launches workers and servers. Each worker loads the snapshot of the topic assignments from HDFS and rebuilds $\mathbf{C_w}$ and $\mathbf{C_d}$. Then the values of $\mathbf{C_w}$ will be pushed to servers and the training can be restarted.

dataset	#docs	#words	#tokens	average doc length	sparse text size
NYTimes	300K	102660	99M	331	1GB
PubMED	8.2M	141044	737M	90	4GB
Tencent	2.5M	88916	1.26B	504	5.6GB
BaiKe	2.8M	98234	418M	148	2.3GB
Production1	159M	3759618	19B	122	90GB
Production2	507M	5588297	60B	119	270GB
Production3	2.2B	3911813	308B	138	1.8TB

Table 1: Dataset Statistics

5.4 Other Details

Aggregation Functions. We implement aggregation methods for parameters to facilitate other computation operations. One example is to calculate the summation of log likelihood value for words. This calculation requires traversing each non-zero value of $\mathbf{C_w}$ and deriving the summation of $logGamma(C_{wk}+\beta) - logGamma(\beta)$. Since $\mathbf{C_w}$ is stored on different servers, LDA* enables the acquisition of results by conducting the aggregation operations on servers at the pull phase to avoid extra data fetching.

Network transfer control. To avoid OOM exception on servers , we impose size limitations to the *pull* and *push* methods at the worker side. First, we put a constraint on the overall size of network requests. Second, we limit the total number of partition requests for each server. The size of each partition is maintained at the worker side to obtain a precise estimation for the response message of each *pull* operation. The value in this size cache is periodically updated through the aggregation method described above.

6. EVALUATION

We evaluate LDA* and compare it with state-of-the-art systems for LDA. We also validate that both of our technical contributions, including the hybrid sampler and asymmetric parameter server architecture, significantly improve the performance of our system.

6.1 Experimental Setup

Datasets. Table 1 summarizes the datasets we used. We use smaller datasets, including NYTimes, PubMED, Tencent, and BaiKe, to evaluate the speed of samplers in a single machine and compare the performance of different systems using larger datasets, including Production1, Production2, Production3, to evaluate the effects of LDA* in the production cluster. Compared with the other three datasets, BaiKe is a dataset with many long documents as well as many short documents. NYTimes and PubMED are public datasets[3], while the other datasets are from our industry partner.

Baseline Systems. We compare our system with Petuum [24] and LightLDA [27]. Petuum is a distributed machine learning system with parameter server architecture. It employs AliasLDA as its sampler at each worker. The distributed implementation of LightLDA also uses a standard parameter server architecture.

Experiment Setting. We conduct distributed experiments on a shared cluster provided by our industry partner with 4600 machines, where each machine has a 2.2GHz CPU with 12 cores, 64GB memory, and 12 × 2TB SATA hard disks. Most of the machines are connected with 10Gbps network while a fraction of them are connected with 1Gbps network. For the distributed settings, the size of JVM heap for worker is 8GB, while it is 4GB for server. Following prior arts, we set the hyperparameters of LDA $\alpha = \frac{50.0}{K}$ and $\beta = 0.01$ for all the following experiments.

[3] https://archive.ics.uci.edu/ml/machine-learning-databases/bag-of-words/

We also compare with existing systems on a smaller-scale cluster, such that all systems can scale. This cluster contains 20 machines, each of which has a 2.2GHz CPU with 16 cores, 64GB memory and 12 × 2TB SATA hard disks.

Metrics. We measure the quality of an LDA model by the log joint likelihood, which can be calculated by

$$L = log\ p(W,Z|\alpha,\beta) = log \prod_d [\frac{\Gamma(\overline{\alpha})}{\Gamma(\overline{\alpha}+L_d)} \prod_k \frac{\Gamma(\alpha_k + C_{dk})}{\Gamma(\alpha_k)}] \prod_k [\frac{\Gamma(\overline{\beta})}{\Gamma(\overline{\beta}+C_k)} \prod_w \frac{\Gamma(\beta + C_{wk})}{\Gamma(\beta)}]$$

We count the wall-clock time for a method to reach a given loglikelihood value in order to compare their performance.

6.2 End-to-End Comparison

We validate that, when comparing end-to-end performance, LDA* is significantly faster than Petuum and LightLDA. We run over the PubMED and Tencent datasets and set $K = 1,000$. We use 20 workers for all systems.

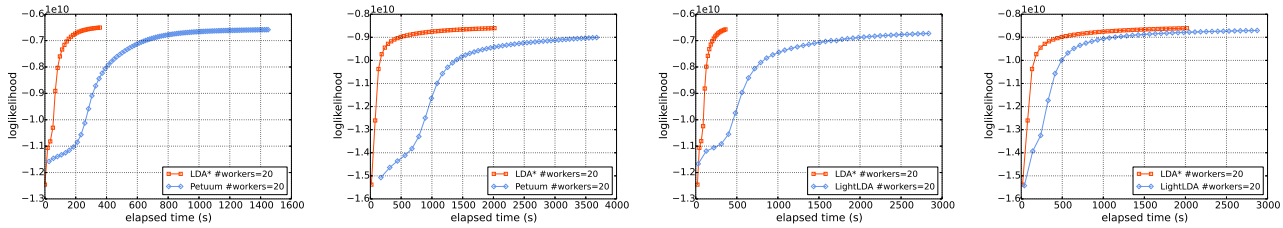
Compared with Petuum. Figures 8(a), 8(b) show the performance comparison between LDA* and Petuum. On the PubMED dataset, to reach the log likelihood value of $-6.6e9$, LDA* requires about 250 seconds, while Petuum spends 1240 seconds. Therefore, LDA* is about 5× faster than Petuum. There are two reasons that contribute to the improvement. The first one is our hybrid sampler, which can outperform AliasLDA on the PubMED dataset. The second one is that Petuum incurs a global synchronization operation after each iteration, while our system avoids this global barrier by partitioning it into many small synchronization units. For the Tencent dataset, our system is about 9× faster than Petuum since this dataset contains lots of long documents and AliasLDA performs poorly on it.

Compared with LightLDA. Figures 8(c), 8(d) present the comparison between LDA* and LightLDA. On the PubMED dataset, to achieve the log likelihood value of $-6.72e9$, LDA* needs 272 seconds, while LightLDA needs 2835 seconds. Our system can be 10× faster than LightLDA in this evaluation. Our hybrid sampler contributes to 5× of the speed increase (refer to the second row of Table 2 for the evaluation of the contribution). For the Tencent dataset, our system can still be 2.6× faster than LightLDA. The main reason for the degradation of improvement is that the hybrid sampler presents similar performance with LightLDA over this dataset. Thus, the improvement is mostly contributed by our distributed architecture and the careful system implementations.

Lesion Study. To understand the impact of our contribution, we compare our system with a variant the disable all optimizations we proposed in this paper. This allows us to make a breakdown for the performance improvement of LDA*. According to the results of Section 6.4.1 and 6.4.3, our two main techniques both make contributions to the performance improvement. Our hybrid samplers can result a 2×-5× speed up on different datasets and our architecture can also obtain a 2×-5× speed up under different settings. When comparing with other systems, the improvement can be larger mainly due to our careful engineering efforts.

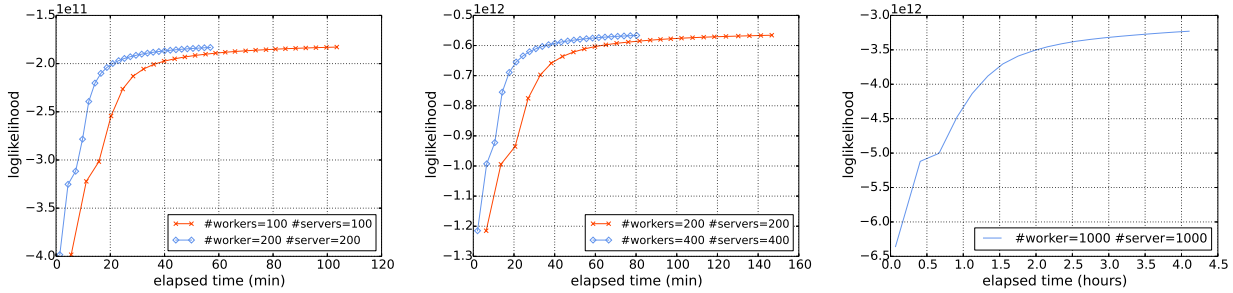
6.3 Results on Production Cluster

We now present the performance results of running LDA* in a real production cluster to handle large datasets. We use Production1, Production2, and Production3 to evaluate the scalability of



(a) vs Petuum (PubMED) (b) vs Petuum (Tencent) (c) vs LightLDA (PubMED) (d) vs LightLDA (Tencent)

Figure 8: Comparing LDA* with Petuum and LightLDA



(a) Production1 K=8k (b) Production2 K=8k (c) Production3 K=8k

Figure 9: Results with three large datasets on Real Production Cluster.

dataset	K& L	F+	Alias	Light	Warp	Hybrid
PubMED	1k&-6.6e9	2489	2865	11024	4400	2322
	8k&-6.7e9	2851	3355	13045	7389	2212
	16k&-6.8e9	3115	3580	15426	8957	2713
NYTimes	1k&-9.7e8	712	879	1600	750	431
	8k&-1e9	706	781	1760	761	592
	16k&-1e9	589	709	1377	554	403
Tencent	1k&-9e9	3500	5108	3564	3087	2142
	8k&-9e9	7746	11839	7500	4747	4341
	16k&-9e9	9910	14072	7800	5894	5773
BaiKe	8k&-3.6e9	1893	2505	3308	2045	1158
	16k&-3.6e9	2763	3065	6930	3792	1779

Table 2: Wall-clock time (seconds) for samplers to reach the same log likelihood.

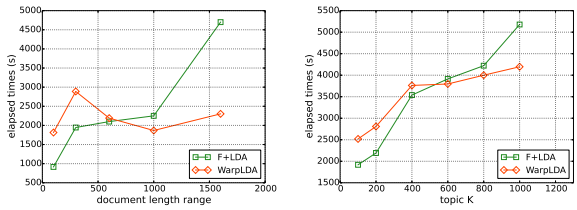
LDA*. We fail to install both Petuum and LightLDA on the production cluster, and therefore, we omit the comparison between them. Figure 9 shows the results. The topic number for all these experiments is set to $8,000$ and we run 100 iterations for all datasets. The JVM heap size is set to 8 GB for each worker and 4 GB for each server, while the direct memory size for each is set to 2 GB.

Figure 9(a) shows the speed on Production1. We can see that LDA* can complete 100 iterations in about 110 minutes with 100 workers. When using more (200) workers, it further decreases to 58 minutes—a near linear speed up. Similar results can be seen in Figure 9(b). One reason for the scalability of LDA* is that the network waiting time is siginificantly reduced by our synchronization method. Moreover, the skew-aware partitioning strategy generates balanced results which avoids the communication bottleneck.

Figure 9(c) shows the performance when training Production3 dataset. We use 1,000 workers and 1,000 servers to scale the training for this dataset, and it can complete in about 4 hours. These experiments demonstrate that LDA* can train large datasets with 1,000 workers in a real production environment. Moreover, we can obtain convergence in hours to satisfy the requirements of workloads in real world.

6.4 System Tradeoffs

We now validate that each of our contributions has significant impact on our end-to-end performance.



(a) Tradeoff on L_d (b) Tradeoff on K

Figure 10: Tradeoff point of L_d and K

6.4.1 *Hybrid Samplers*

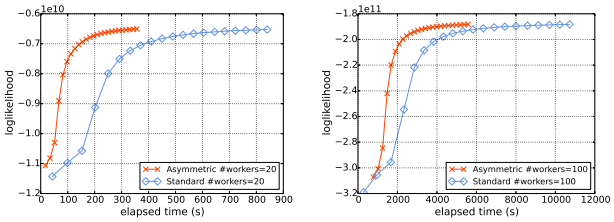
Table 2 gives a summary of the performance for all samplers by presenting their wall-clock time to a given value of loglikelihood.

We see that our hybrid sampler outperforms the fastest existing samplers on all datasets. On NYTimes and BaiKe, which contains both short documents and long documents, our sampler can outperform both F+LDA and WarpLDA by 1.7-2.1×. This is because our sampler is able to choose the most appropriate sampler to use. For PubMED dataset, our hybrid sampler is 2-3× faster than WarpLDA, while having comparable performance to F+LDA. For Tencent dataset, our hybrid sampler is 1.7× faster than WarpLDA and is comparable with WarpLDA. For the detailed analysis of other samplers, please refer to Section 3.1.

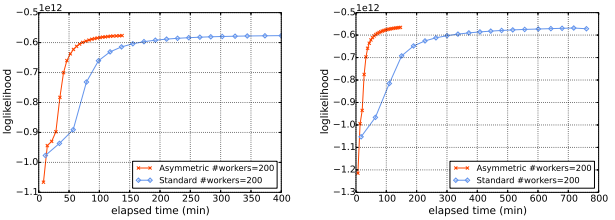
6.4.2 *Tradeoff of K and L_d*

We now validate that both L_d and K form a tradeoff between SA and MH samplers.

Tradeoff of L_d. To determine the tradeoff point of L_d, we construct five datasets with different document lengths. The first dataset consists of documents with $L_d \in [0, 200)$, while the second one contains documents with $L_d \in [200, 400)$. The lengths of documents for the other three datasets are $[400, 600)$, $[600, 1, 200)$, and $[1, 200, 2, 000)$. We run F+LDA and WarpLDA with $K = 8,000$ on these five datasets and their wall-clock time to reach the same convergent loglikelihood are shown in Figure 10(a). From this figure, we can see that on dataset with $L_d \in [0, 200)$, F+LDA is 2× faster than WarpLDA, while F+LDA is 2× slower than WarpLDA on the dataset with $L_d \in [1, 200, 2, 000)$. For the dataset containing documents with $L_d \in [400, 600)$, F+LDA and WarpLDA



(a) PubMED(K=1000) (b) Production1(K=1000)



(c) Production2(K=1000) (d) Production2(K=8000)

Figure 11: Impact of Asymmetric Architecture and skew-aware partitioning

present nearly the same performance. Therefore, in our implementation of the hybrid sampler, we use F+LDA for documents with length less than 600 and choose WarpLDA otherwise.

Tradeoff of K. To determine the tradeoff point of K, we evaluate both F+LDA and WarpLDA on the Tencent dataset with K range from 100 to 1,000. The wall-clock time to reach the same loglikelihood value is presented in Figure 10(b). We can see that F+LDA can outperform WarpLDA when K is small, while WarpLDA is faster when K becomes larger. The cross point happens when $K \simeq 600$ in our experiments. Therefore, in our implementation of the hybrid sampler, we use F+LDA when K is less than 600; otherwise we use WarpLDA.

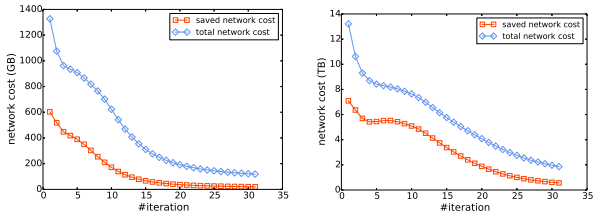
6.4.3 Asymmetric Architecture

Overall Impact. Figure 11 shows the overall performance of three different datasets, with and without our asymmetric architecture and skew-aware partitioning, respectively. We run both for 100 iterations and set $K = 1000$. We also set $K = 8000$ for Production2. We see that on both PubMED and Production1, even with only 20 machines, an asymmetric architecture and the skew-aware partitioning strategy speed up our system by 2×. For the Production2 dataset, we can obtain about 3× performance speed up when $K = 1000$ since communication cost dominates the overall performance when training with more words and more workers. When K is set to 8000, the performance improvement can be increased to 5 times since larger K results in more communication costs.

Communication cost. Figure 12 shows the effects of the asymmetric architecture on the communication cost. We use both Production2 and Production3 to conduct this evaluation. The partition number for Production2 (resp. Production3) is set to 300 (resp. 1,000). On Production2, an asymmetric architecture can reduce about 45% of the communication cost for the first iteration and 34% of costs for the first 30 iterations. On Production3, an asymmetric architecture can save about 55% of the communication cost for the first 30 iterations. Although the improvement decreased along with the iterations—due to the decreased size of $\mathbf{C_w}$— the reduced cost for the first dozens of iterations can help improve the performance.

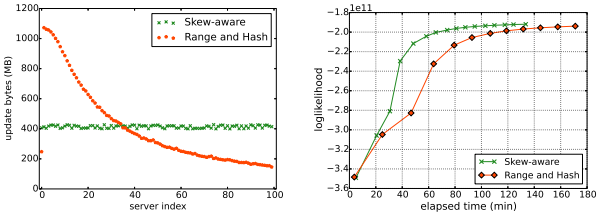
Effects of Partition Method. We now run experiments [4] on Production1 with 100 workers and 100 servers to validate the impact of our skew-aware partitioning method. We partition $\mathbf{C_w}$ into 3,000 partitions and assign them to servers. Figure 13(a) shows the amount of data that gets pushed at the first iteration for two different partition methods: (1) The *Range and Hash* method first partitions $\mathbf{C_w}$ with range partitioning, using a balanced word count in each partition, and then hashes each partition to servers; and (2) The *Skew-Aware* method partitions the $\mathbf{C_w}$ as described in Section 4.2. We see that the skew-aware method can generate balanced partition results for each server. Figure 13(b) presents the convergence rate under these two partition methods. The skew-aware method can be about 50% faster.

[4] When running with other two large datasets, the *Range and Hash* method always fails due to the OOM exception on the workhorse server.



(a) Production2 (b) Production3

Figure 12: Reduced communication cost



(a) pushing request size (b) convergence speed

Figure 13: Effects of skew-aware partitioning.

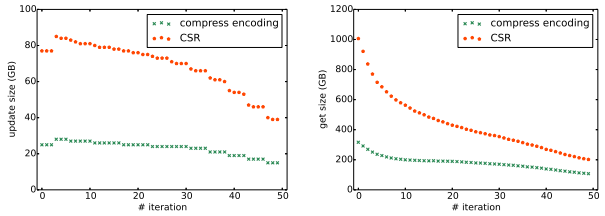
6.4.4 Effects of Compressed Encoding

We now validate the impact of the compressed encoding method described in Section 5.1. We conduct the experiments on Production2 with 300 workers and 300 servers. We measure the total message size over all workers at each iteration. We compare our method with the CSR format because the pulling response and the pushing requests are sparse.

Figure 14(a) and Figure 14(b) show the amount of network messages through pulling and pushing in each iteration respectively. We see that our encoding method can reduce the communication cost by 2× to 4×. For the pushing request, our encoding method can decrease the network communication from 80GB to 23GB at the first 20 iterations. For the pulling operation, we can save more than 500GB data at the start of training. Therefore, we can dramatically reduce the communication cost through this method, which saves a massive amount of communication cost and guarantees the successful running of each worker and server.

7. RELATED WORK

Samplers for LDA. There are a range of Gibbs samplers that have been proposed to reduce the sampling complexity of LDA. SparseLDA [25] is the first to take advantage of the sparse structure of $\mathbf{C_d}$ and $\mathbf{C_w}$ to achieve $O(K_d + K_w)$ sampling complexity. However, for a large dataset, the value of K_w for popular words equals to K—AliasLDA [11] and F+LDA [26] further reduces the complexity to $O(K_d)$ by factorizing the posterior distribution. LightLDA [27] proposes an $O(1)$ method by alternately sampling from two simple proposals through MH method, while WarpLDA [2] reorders the sampling operations from these two proposals to reduce random memory access. Although the complexity for one sampling operation is $O(1)$ for them, we find that they need



(a) summation of pushing size (b) summation of pulling size

Figure 14: Effects of encoding method on Production2.

more sampling operations to generate one sample, since the MH method requires mix time. FastLDA [18] uses the skew property of Eq. 1 to calculate only a fraction of the K topic probabilities per sampling operation.

Those samplers are all designed for a specific tradeoff space which can lead to suboptimal solutions. We systematically study the tradeoff space for Gibbs sampling algorithm and build a hybrid sampler, which employs an automatic optimizer to choose one from them according to the length of document and the value of topic. To our best knowledge, our study is the first one which explores the tradeoff space and builds a novel sampler to acchieve speed up.

Distributed LDA. To scale up the training of LDA for large datasets, different researchers have proposed different parallelization methods. All parallelization methods partition documents into multiple partitions and each worker conducts sampling operations. AD-LDA [15] proposes an approximate sampling method where each worker uses the stale version of the $\mathbf{C_w}$ matrix and requires synchronization among all workers after each iteration. PLDA [23] is an MPI and MapReduce implementation of AD-LDA. PLDA+ [13] proposes to sample over word order at each worker and then utilize the pipeline method to overlap the computation and communication. YahooLDA [19] proposes a distributed key-value cache to share the $\mathbf{C_w}$ matrix among different workers. However, it samples tokens through document order and requires maintenance of one copy of $\mathbf{C_w}$ at each worker. NomadLDA [26] utilizes the nomad token to perform asynchronous sampling while guaranteeing there will be no update conflicts. Instead of using the nomad token, Petuum [8] employs a central scheduler to coordinate the updates. ParameterServer [12] employs 6,000 machines to train LDA with 5 million distinct words and 2000 topics. However, it still takes more than 20 hours to reach convergence, while LDA* can obtain convergence in about 4 hours for 8,000 topics.

Other Inference Algorithms. In this paper, we focus on sampling-based approaches such as Gibbs sampling and Metropolis-Hastings. Apart from these approaches, Variational Inference is another popular algorithm [21, 1], which finds a distribution to approximate the posterior distribution of LDA. Although Collapsed Variational Inference can be faster than traditional CGS algorithm, it is slower than recent state-of-the-art samplers. Moreover, there is no research on how to utilize the sparse property to reduce memory cost—it requires storage of an array of length K for each token. Stochastic learning method is another alternative to provide faster convergence speed and lower memory cost. Related studies include stochastic variational inference (SVI) [7], stochastic collapsed variational inference [3], and stochastic gradient Riemann Langevin dynamics (SGRLD) [16]. We hope our study can be used to speed up these variational and stochastic algorithms in the future.

8. CONCLUSION

In this paper, we systematically studied the state-of-the-art samplers for LDA and discoverd a tradeoff between the Sparse-Aware sampler and Metropolis-Hastings samplers. Based on this tradeoff, we developed a hybrid sampler that employs different samplers for documents of different lengths. We further proposed an asymmetric parameter server to achieve scalability in the distributed environment. We built a system, named LDA*, to accelerate and scale the training of LDA for large datasets in real-world clusters. LDA* has been deployed in Tencent for various workloads of topic modeling.

9. REFERENCES

[1] A. Asuncion, M. Welling, P. Smyth, and Y. W. Teh. On smoothing and inference for topic models. In *AUAI*, pages 27–34, 2009.

[2] J. Chen, K. Li, J. Zhu, and W. Chen. Warplda: a cache efficient o (1) algorithm for latent dirichlet allocation.

[3] J. Foulds, L. Boyles, C. DuBois, P. Smyth, and M. Welling. Stochastic collapsed variational bayesian inference for latent dirichlet allocation. In *SIGKDD'2013*.

[4] T. L. Griffiths and M. Steyvers. Finding scientific topics. *PNAS*, 101(suppl 1):5228–5235, 2004.

[5] M. Harvey, F. Crestani, and M. J. Carman. Building user profiles from topic models for personalised search. In *CIKM*, pages 2309–2314. ACM, 2013.

[6] Q. Ho, J. Cipar, H. Cui, S. Lee, J. K. Kim, P. B. Gibbons, G. A. Gibson, G. Ganger, and E. P. Xing. More effective distributed ml via a stale synchronous parallel parameter server. In *NIPS*, pages 1223–1231, 2013.

[7] M. D. Hoffman, D. M. Blei, C. Wang, and J. W. Paisley. Stochastic variational inference. *JMLR*, 14(1):1303–1347, 2013.

[8] J. K. Kim, Q. Ho, S. Lee, X. Zheng, W. Dai, G. A. Gibson, and E. P. Xing. Strads: a distributed framework for scheduled model parallel machine learning. In *EuroSys*, page 5. ACM, 2016.

[9] Y. LeCun, Y. Bengio, and G. Hinton. *Nature*, 2015.

[10] D. A. Levin, Y. Peres, and E. L. Wilmer. *Markov chains and mixing times*. American Mathematical Soc.

[11] A. Q. Li, A. Ahmed, S. Ravi, and A. J. Smola. Reducing the sampling complexity of topic models. In *SIGKDD*, pages 891–900. ACM, 2014.

[12] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *Proc. OSDI*, pages 583–598, 2014.

[13] Z. Liu, Y. Zhang, E. Y. Chang, and M. Sun. Plda+: Parallel latent dirichlet allocation with data placement and pipeline processing. *TIST*, 2(3):26, 2011.

[14] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013.

[15] D. Newman, P. Smyth, M. Welling, and A. U. Asuncion. Distributed inference for latent dirichlet allocation. In *NIPS*, pages 1081–1088, 2007.

[16] S. Patterson and Y. W. Teh. Stochastic gradient riemannian langevin dynamics on the probability simplex. In *NIPS*, pages 3102–3110, 2013.

[17] J. Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1988.

[18] I. Porteous, D. Newman, A. Ihler, A. Asuncion, P. Smyth, and M. Welling. Fast collapsed gibbs sampling for latent dirichlet allocation. In *SIGKDD*, pages 569–577. ACM, 2008.

[19] A. Smola and S. Narayanamurthy. An architecture for parallel topic models. *PVLDB*, 3(1-2):703–710, 2010.

[20] D. Suciu, D. Olteanu, C. Ré, and C. Koch. *Probabilistic Databases*. Synthesis Lectures on Data Management. Morgan & Claypool Publishers, 2011.

[21] Y. W. Teh, D. Newman, and M. Welling. A collapsed variational bayesian inference algorithm for latent dirichlet allocation. In *NIPS*, pages 1353–1360, 2006.

[22] A. J. Walker. An efficient method for generating discrete random variables with general distributions. *TOMS*, 3(3):253–256, 1977.

[23] Y. Wang, H. Bai, M. Stanton, W.-Y. Chen, and E. Y. Chang. Plda: Parallel latent dirichlet allocation for large-scale applications. In *AAIM*, pages 301–314. Springer, 2009.

[24] E. P. Xing, Q. Ho, W. Dai, J. K. Kim, J. Wei, S. Lee, X. Zheng, P. Xie, A. Kumar, and Y. Yu. Petuum: A new platform for distributed machine learning on big data. *arXiv preprint arXiv:1312.7651*, 2013.

[25] L. Yao, D. Mimno, and A. McCallum. Efficient methods for topic model inference on streaming document collections. In *SIGKDD*, pages 937–946. ACM, 2009.

[26] H.-F. Yu, C.-J. Hsieh, H. Yun, S. Vishwanathan, and I. S. Dhillon. A scalable asynchronous distributed algorithm for topic modeling. In *WWW*, pages 1340–1350. ACM, 2015.

[27] J. Yuan, F. Gao, Q. Ho, W. Dai, J. Wei, X. Zheng, E. P. Xing, T.-Y. Liu, and W.-Y. Ma. Lightlda: Big topic models on modest computer clusters. In *WWW*, pages 1351–1361. ACM, 2015.

[28] C. Zhang. *DeepDive: A Data Management System for Automatic Knowledge Base Construction*. PhD thesis, University of Wisconsin-Madison, 2015.

[29] J. Zhu, A. Ahmed, and E. P. Xing. Medlda: maximum margin supervised topic models. *Journal of Machine Learning Research*, 13(Aug):2237–2278, 2012.