LightLDA: Big Topic Models on Modest Computer Clusters

Jinhui Yuan[1], Fei Gao[1],
Qirong Ho[3], Wei Dai[2], Jinliang Wei[2], Xun Zheng[2], Eric P. Xing[2],
Tie-Yan Liu[1], and Wei-Ying Ma[1]

[1]Microsoft Research
[2]School of Computer Science, Carnegie Mellon University
[3]Institute for Infocomm Research, A*STAR, Singapore

{jiyuan,feiga,tie-yan.liu,wyma}@microsoft.com,hoqirong@gmail.com,{wdai,jinlianw,xunzheng,epxing}@cs.cmu.edu

ABSTRACT

When building large-scale machine learning (ML) programs, such as massive topic models or deep neural networks with up to trillions of parameters and training examples, one usually assumes that such massive tasks can only be attempted with industrial-sized clusters with thousands of nodes, which are out of reach for most practitioners and academic researchers. We consider this challenge in the context of topic modeling on web-scale corpora, and show that with a modest cluster of as few as 8 machines, we can train a topic model with 1 million topics and a 1-million-word vocabulary (for a total of 1 trillion parameters), on a document collection with 200 billion tokens — a scale not yet reported even with thousands of machines. Our major contributions include: 1) a new, highly-efficient $\mathcal{O}(1)$ Metropolis-Hastings sampling algorithm, whose running cost is (surprisingly) agnostic of model size, and empirically converges nearly an order of magnitude more quickly than current state-of-the-art Gibbs samplers; 2) a model-scheduling scheme to handle the big model challenge, where each worker machine schedules the fetch/use of sub-models as needed, resulting in a frugal use of limited memory capacity and network bandwidth; 3) a differential data-structure for model storage, which uses separate data structures for high- and low-frequency words to allow extremely large models to fit in memory, while maintaining high inference speed. These contributions are built on top of the Petuum open-source distributed ML framework, and we provide experimental evidence showing how this development puts massive data and models within reach on a small cluster, while still enjoying proportional time cost reductions with increasing cluster size.

Categories and Subject Descriptors

H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed Systems*; G.3 [**Probability and Statistics**]: Probabilistic algorithms (including Monte Carlo); G.4 [**Mathematical Software**]: Parallel and vector implementations

General Terms

Design, Algorithms, Experimentation, Performance

Copyright is held by the International World Wide Web Conference Committee (IW3C2). IW3C2 reserves the right to provide a hyperlink to the author's site if the Material is used in electronic media.
WWW 2015, May 18–22, 2015, Florence, Italy.
ACM ACM 978-1-4503-3469-3/15/05.
http://dx.doi.org/10.1145/2736277.2741115 .

Keywords

Large Scale Machine Learning; Distributed Systems; Topic Model; Model Scheduling; Data Parallelism; Metropolis-Hastings; Parameter Server; Petuum

1. INTRODUCTION

Topic models (TM) are a popular and important modern machine learning technology that has been widely used in text mining, network analysis and genetics, and more other domains [17, 5, 23, 24, 2]. Their impact on the technology sector and the Internet has been tremendous — numerous companies having developed their large-scale TM implementations [13, 1, 21], with applications to advertising and recommender systems. A key goal of contemporary research is to scale TMs, particularly the Latent Dirichlet Allocation (LDA) model [5], to web-scale corpora (Big Data). Crucially, Internet-scale corpora are significantly more complex than smaller, well-curated document collections, and thus require high-capacity parameter space featuring up to millions of topics and vocabulary words (and hence trillions of parameters, i.e. Big Models), in order to capture long-tail semantic information that would otherwise be lost when learning only a few thousands of topics [21].

To achieve massive data and model scales, a typical approach is to engineer a distributed system that can efficiently execute well-established parallelization strategies (e.g., split documents over workers, which have shared access to all parameters, possibly stored in a distributed manner) for LDA [16, 12, 21], while applying algorithmic speedups such as the SparseLDA [22] or AliasLDA [11] samplers to further decrease running time. Such efforts have enabled LDA models with tens of billions of parameters to be inferred from billions of documents, using up to thousands of machines [13, 1, 12, 21]. While such achievements are impressive, they are unfortunately costly to run: for instance, a cluster of thousands of machines will cost millions of dollars to set up (not to mention the future costs of power and regular maintenance). Alternatively, one might rent equivalent computation capacity from a cloud provider, but given the current typical price of $\geq$ \$1 per hour per research-grade machine, a single month of operations would cost millions of dollars. Neither option is feasible for the majority of researchers and practitioners, who are more likely to only have modest budgets.

Rather than insisting that big computing is the only way to solve large ML problems, what if we could solve large topic models with trillions of parameters in a more cost-effective manner, by providing an implementation that is efficient enough for modest clusters with at most tens of machines? We approach this problem at three levels: (1) We perform distributed cluster LDA inference in a *data-*

parallelism and *model-scheduling* fashion: in each iteration, every worker processes a subset of the training data, scheduling the fetch/use of a sub-model slice by slice. We do this in a memory- and network-efficient manner: we fix each data partition to a particular worker (i.e., data are never moved around), but we regularly change which model partitions each machine works on — specifically, we store the model in a distributed parameter server, and stream model partitions as needed to worker machines; (2) we develop a novel Metropolis-Hastings (MH) sampler with carefully constructed proposals that allows for $\mathcal{O}(1)$ amortized sampling time per word/token, resulting in a fast convergence (in terms of real time) that beats existing state-of-the-art LDA samplers by a significant margin [22, 11]; (3) we employ a differential data structure to leverage the fact that web-scale corpora exhibit both high-frequency as well as low-frequency words, which can be treated differently in storage (i.e., a hybrid internal representation). This gives a best-of-both-worlds outcome: the high memory efficiency of sparse data structures, with the high performance of dense data structures.

By realizing these ideas using the open-source Petuum framework (www.petuum.org) [9, 10, 6], we have produced a compute-and-memory efficient distributed LDA implementation, LightLDA, that can learn an LDA model with one trillion model parameters (one million topics by one million vocabulary words) from billions of documents (200 billion tokens in total), on a computer cluster with as few as 8 standard machines (whose configuration is roughly similar to a typical compute instance from a cloud provider) in 180 hours, which proportionally drops to 60 hours on 24 machines. In terms of parameter size, our result is two orders of magnitude larger than recently-set LDA records in the literature, which involved models with tens of billions of parameters and typically used massive industrial-scale clusters [12, 21, 1, 13]; our data size is also one order of magnitude larger than those same works[1]. We show that LightLDA converges significantly faster than existing implementations across several datasets and model settings (vocabulary size and number of topics). Notably, LightLDA's per-token computational complexity is independent of model size, hence it enjoys high throughput even under massive models.

Overall, LightLDA benefits both from a highly efficient Metropolis Hastings sampler built on a new proposal scheme, and a highly efficient distributed architecture and implementation built on Petuum. It represents a truly lightweight realization (hence its name LightLDA) of a massive ML program, which we hope will be easily accessible to ordinary users and researchers with modest resources. Compared to using alternative platforms like Spark and Graphlab that also offer highly sophisticated data- or model- parallel systems, or designing bespoke ground-up solutions like PLDA and YahooLDA, we suggest that our intermediate approach that leverages both simple-but-critical algorithmic innovation and lightweight ML-friendly system platforms stands as a highly cost-effective solution to Big ML.

2. CHALLENGES AND RELATED WORK

Before discussing the challenges, we briefly review the Latent Dirichlet Allocation (LDA) [5] model to establish nomenclature. Specifically, LDA assumes the following generative process for each document in a corpus:

- $\varphi_k \sim$ Dirichlet(β): Draw word distribution φ_k per topic k.
- $\theta_d \sim$ Dirichlet(α): Draw topic distribution θ_d per document d.
- $n_d \sim$ Poisson(γ): For each document d, draw its length n_d (i.e., the number of tokens it contains).
- For each token $i \in \{1, 2, \ldots, n_d\}$ in document d:
  - $z_{di} \sim$ Multinomial(θ_{di}): Draw the token's topic.
  - $w_{di} \sim$ Multinomial($\varphi_{z_{di}}$): Draw the token's word.

[1][21] used 4.5 billion tokens, while [12] used 5 billion short documents of unspecified length.

To find the most plausible topics in a corpus and document-topic assignments, one must infer the posterior distribution of latent variables in LDA model, by using either a variational- or sampling-based inference algorithm. Sampling-based algorithms are known to yield very sparse updates that make them well-suited to settings with a massive number of topics and distributed implementation. In particular, the collapsed Gibbs sampler for LDA [7] is preferred. It works as follows: all variables except the token's topic indicator z_{di} are analytically integrated out, and we only need to Gibbs sample z_{di} according to

$$p(z_{di} = k|\text{rest}) \propto \frac{(n_{kd}^{-di} + \alpha_k)(n_{kw}^{-di} + \beta_w)}{n_k^{-di} + \bar{\beta}}, \qquad (1)$$

where z_{di} follows a Multinomial distribution with K outcomes (i.e., K is the number of topics in the model), w is short for w_{di}, $\bar{\beta} := \sum_w \beta_w$, n_{kd}^{-di} is the number of tokens in document d that are assigned to topic k (excluding z_{di}), n_{kw}^{-di} is the number of tokens with word w (across all documents) that are assigned to topic k (excluding z_{di}), and n_k^{-di} is the number of tokens (across all docs) assigned to topic k (excluding z_{di}). To avoid costly recalculation, these counts (also called "sufficient statistics") are cached as tables, and updated whenever a token topic indicator z_{di} changes. In particular, the set of all counts n_{kd} is colloquially referred to as the *document-topic table* (and serves as the sufficient statistics for θ_d), while the set of all counts n_{kw} is known as the *word-topic table* (and forms the sufficient statistics for φ_k).

While training LDA model, the Gibbs sampler needs to scan the whole corpus for hundreds of times (i.e., iterations). In each iteration, it sequentially process all the tokens in the corpus according to Eq. 1 with an $\mathcal{O}(K)$ per-token complexity. Therefore, the amount of required computation for training LDA with Gibbs sampling depends on both the scale of corpus (specifically, the number of tokens) and the size of model (the number of topics). However, in many real-word applications especially the web-scale corpora, the problem exhibits billions of documents and thousands (even millions) of topics [21]. Much research has been invested on scaling LDA to ever-larger data and model sizes; existing papers usually show an algorithmic focus (i.e. better LDA inference algorithm speed) or a systems focus (i.e. better software to execute LDA inference on a distributed cluster) — or even both foci at once.

In any case, the inference algorithm is one limiting factor to efficiency. For example, the Peacock system [21] adopts the SparseLDA inference algorithm [22], which has a lower per-token computational complexity than the standard collapsed Gibbs sampler. We note that the recently developed AliasLDA algorithm [11] provides a further improvement on SparseLDA sampler. However, the per-token computational complexity of AliasLDA still depends on the document length and the sparsity of document-topic distribution, therefore in some settings, AliasLDA may not be a substantial improvement over SparseLDA. To tackle this challenge, we develop a new $\mathcal{O}(1)$-per-token Metropolis-Hastings sampler that is nearly an order of magnitude faster than the existing algorithms — which allows us to process the same quantity of data with fewer resources in a reasonable amount of time.

To tackle the Big Data and the Big Model challenges, algorithms alone are not enough, and we must also resort to an efficient distributed implementation. Recent large-scale implementations of LDA [13, 1, 21, 12] demonstrate that training is feasible on big document corpora (up to billions of documents) using large, industrial-scale clusters with thousands to tens of thousands of CPU cores. At a high level, the above large-scale LDA papers differ substantially in the degree to which they employ data-parallelism

(commonly realized by splitting documents over machines) versus model-parallelism (often realized by splitting *word-topic* distributions over machines). Data-parallelism is a well-accepted solution to the Big Data problem, in which many distributed CPU cores can be used to sample thousands of tokens simultaneously. Furthermore, the total pool of memory across the cluster allows the entire (large) model to be persisted in a distributed fashion, hence supporting in-memory computation. This strategy addresses a key issue in the Big Model problem. Our distributed LDA implementation is based on a parameter server (PS) architecture [1, 9, 12]. PS architectures have shown appealing properties for distributed machine learning (e.g., allowing asynchronous communication in data-parallelism), and we use Petuum's PS for its theoretically-guaranteed SSP consistency model [9]. However, instead of using clusters with thousands of nodes, this paper aims to solve the Big Data and Big Model challenges with a modest computer cluster (i.e., tens of nodes).

In implementations that are predominately data-parallel [1, 12], the strategy is to make the *word-topic* distributions globally-shared, so that the inference algorithm can be agnostic to their physical layout across machines. The training data are partitioned and distributed to worker machines — that is, each worker only processes a subset of data — and the system schedules the inference computation on token topic indicators z_{di} in a document-centric manner. We do not expect either [1] or [12] to handle very large topic models with over 1 trillion parameters (the largest reported result was 10 billion parameters in [12]). In particular, [12] depends on the assumption that once the entire corpus has been distributed to sufficiently many machines, the local documents on each machine will only activate a small portion of the complete model, and therefore the memory required by each machine will not be too large. Consequently, their design cannot handle large topic models without a large computer cluster. The RLU cache technique used in an earlier version of Petuum [9] can partially resolve this issue and allow big models with data-parallel computing in small computer cluster: when sub-models corresponding to frequent words are accessed, they are likely to be in the local cache. However, for sub-models corresponding to long-tail words, the cache miss rate will be high, and a lot of network communication will be triggered.

The Peacock system [21] realizes both data- and model-parallel ideas, by grouping token topic indicators z_{di} according to their words w_{di}; this is beneficial because it reduces the proportion of the *word-topic* distributions that must be held at each worker machine. In particular, [21] adopted a grid-like model-parallel partitioning strategy, where each worker handles a subset of data and a part of the model. A shortcoming of this particular strategy is that workers only process one part of any particular document, so each document must be processed by several worker machines. Hence the system not only needs to synchronize the word-topic distributions among workers, but also the document-topic distributions — in other words, both training data and model need to be transferred among workers. In typical clusters, network bandwidth is often the largest bottleneck in the whole system, and the additional network overhead due to transmitting training data and doc-topic distributions will reduce efficiency.

3. A MODEL-SCHEDULING SCHEME FOR BIG MODEL

To tackle the aforementioned problems in classic parallelization schemes, we proposed a new scheme called *data-parallelism and model-scheduling*. As the name suggests, our scheme employs data-parallelism, by partitioning and distributing the training data into different worker machines; this ensures that each document

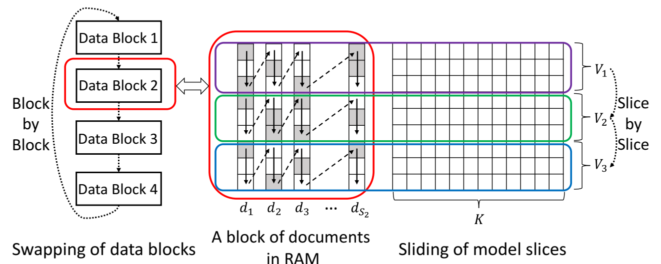


Figure 1: Model scheduling in each worker machine. The arrows across documents $d_1, d_2, \ldots$ indicate the token sampling order — observe that we sample tokens z associated with words v in the *word-topic table* (model) slice V_1, before moving on to tokens corresponding to V_2, and so forth. Once all document tokens in the data block have been sampled, the system loads the next data block from disk. Each document is sparse (with respect to the vocabulary): shaded cells indicate that the document has one or more tokens z corresponding to word v, whereas white indicate that the document has no tokens corresponding to word v (and are hence skipped over).

will be processed by one fixed worker, so training data and doc-topic distributions do not have to be synchronized among workers. Model scheduling refers to how we handle the distributed (big) model or word-topic table; we take extra steps to maximize memory and CPU efficiency. The basic idea is to split the *word-topic* distributions (the LDA model) into slices, and fetch/use the model from the parameter server slice-by-slice only when needed (like streaming). This new distribution scheme is more communication-efficient than the implementations of LDA mentioned in Section 2.

At minimum, any distributed LDA implementation must partition (1) the token topic indicators z_{di} and doc-topic table n_{kd} (collectively referred to as the data), as well as (2) the *word-topic table* n_{kw} (the model). When an LDA sampler is sampling a token topic indicator z_{di}, it needs to see the specific row $n_{kw_{di}}$ in the *word-topic table* (as well as the complete document d). However, naive partitioning can lead to situations where some machines touch a large fraction of the *word-topic table*: suppose we sampled every document's tokens in sequence, then the worker would need to see all the rows in *word-topic table* corresponding to words in the document. Using our fast Metropolis-Hastings sampler (described in the following section), each worker machine can sample thousands of documents per second (assuming hundreds of tokens per document); furthermore, we have empirically observed that a few million documents (out of billions in our web-scale corpus) is sufficient to activate almost the entire *word-topic table*. Thus, the naive sequence just described would rapidly swap parts of the *word-topic table* (which could be terabytes in size) in and out of each worker's RAM, generating a prohibitive amount of network communication.

The model-scheduling scheme proposed in this paper is meant to resolve the conflict between fast LDA sampling and limited memory capacity at each worker. The data partition assigned to a particular worker machine is most likely to be too big to reside in the worker machine's RAM. Therefore, the private data of each worker is further split into a few blocks. While generating the data blocks prior to running the LDA sampler, and we note that it is cheap to determine which vocabulary words are instantiated by each block. This information is attached as meta-data to the block. As shown in Figure 1, when we load a data block (and its meta-data) into local memory (denoted by the red rectangle), we choose a small set of words (say V_1 in the figure) from the block's local words. The set of words is small enough that the corresponding rows $n_{\cdot,w_{di}}$ in *word-topic table* can be held in the worker machine's local memory – we call this set of rows a "model slice". The worker fetches the model

slice over the network, and the sampler only samples those tokens in the block that are covered by the fetched slice; all other tokens are not touched. In this manner, each worker only maintains a thin model slice in local memory, and re-uses it for all the documents in the current data block. Once all tokens covered by the slice have been sampled, the worker fetches the next model slice over the network (say V_2), and proceeds to sample the tokens covered by it. In this manner (similar to sliding windows in image processing or the TCP/IP protocol, but managed by a local scheduler), a worker processes all the tokens in a data block, one slice at a time, before finally loading the next block from disk. This swapping of blocks to and from disk is essentially out-of-core execution.

In addition to keeping worker memory requirements low, this model-scheduling also mitigates network communication bottleneck, in the following ways: (1) workers do not move onto the next model slice until all the tokens associated with the current slice have been sampled, hence we do not need to apply caching and eviction strategies to the model (which could incur additional communication, as model slices are repeatedly swapped in and out); (2) we overlap the computation and the loading (data blocks from disk, model slices from a distributed parameter server) with pipelining to hide I/O and network communication latency (further details in Section 6). We note that PLDA+ [13] makes use of pipelining in a similar (though not exactly identical) spirit. Finally, we point out that the model-scheduling strategy "sends the model to the data", rather than the converse. This is motivated by two factors: (1) the data (including tokens w_{di} and corresponding topic indicators z_{di}) are much larger than the model (even when the model may have trillions of parameters); (2) as the model converges, it gets increasingly sparse (thus lowering communication), while the data size remains constant. Other distributed LDA designs either adopt a "send data to model" strategy or have to communicate both data and model among workers [21], which is costly in our opinion.

4. A FAST MCMC ALGORITHM

As just discussed, model scheduling enables very large, trillion-parameter LDA models to be learned from billions of documents even on small clusters. However, it alone does not allow huge LDA models to be trained *quickly* or in acceptable time, and this motivates our biggest technical contribution: a novel sampling algorithm for LDA, which converges significantly faster than recent algorithms such as SparseLDA [22] and AliasLDA [11]. In order to explain our algorithm, we first review the mechanics of SparseLDA and AliasLDA.

SparseLDA. SparseLDA [22] exploits the observation that (1) most documents exhibit a small number of topics, and (2) most words only participate in a few topics. This manifests as *sparsity* in both the *doc-topic* and *word-topic* tables, which SparseLDA exploits by decomposing the collapsed Gibbs sampler conditional probability (Eq. 1) into three terms:

$$p(z_{di}=k|\text{rest}) \propto \underbrace{\frac{\alpha_k \beta_w}{n_k^{-di}+\bar{\beta}}}_{r} + \underbrace{\frac{n_{kd}^{-di}\beta_w}{n_k^{-di}+\bar{\beta}}}_{s} + \underbrace{\frac{n_{kw}^{-di}(n_{kd}^{-di}+\alpha_k)}{n_k^{-di}+\bar{\beta}}}_{t}. \quad (2)$$

When the Gibbs sampler is close to convergence, both the second term s and the third term t will become very sparse (because documents and words settle into a few topics). SparseLDA first samples one of the three terms r, s or t, according to their probability masses summed over all K outcomes. Then, SparseLDA samples the topic k conditioned upon which term r, s or t was chosen. If s or t was chosen, then sampling the topic k takes $\mathcal{O}(K_d)$ or $\mathcal{O}(K_w)$ time respectively, where K_d is the number of topics document d contains, and K_w is the number of topics word w belongs to. The amortized sampling complexity of SparseLDA is $\mathcal{O}(K_d + K_w)$, as opposed to $\mathcal{O}(K)$ for the standard collapsed Gibbs sampler.

AliasLDA. AliasLDA [11] proposes an alternative decomposition to the Gibbs sampling probability:

$$p(z_{di}=k|\text{rest}) \propto \underbrace{\frac{n_{kd}^{-di}(n_{kw}^{-di}+\beta_w)}{n_k^{-di}+\bar{\beta}}}_{u} + \underbrace{\frac{\alpha_k(n_{kw}+\beta_w)}{n_k+\bar{\beta}}}_{v}. \quad (3)$$

AliasLDA pre-computes an alias table [20] for the second term, which allows it to be sampled in $\mathcal{O}(1)$ time via Metropolis-Hastings. By re-using the table over many tokens, the $\mathcal{O}(K)$ cost of building the table is also amortized to $\mathcal{O}(1)$ per token. The first term u is sparse (linear in K_d, the current number of topics in document d), and can be computed in $\mathcal{O}(K_d)$ time.

4.1 Metropolis-Hastings Sampling

SparseLDA and AliasLDA achieve $\mathcal{O}(K_d + K_w)$ and $\mathcal{O}(K_d)$ amortized sampling time per token, respectively. Such accelerated sampling is important, because we simply cannot afford to sample token topic indicators z_{di} naively; the original collapsed Gibbs sampler (Eq. 1) requires $\mathcal{O}(K)$ computation per token, which is clearly intractable at $K = 1$ million topics. SparseLDA reduces the sampling complexity by exploiting the sparsity of problem, while AliasLDA harnesses the alias approach together with the Metropolis-Hastings algorithm [15, 8, 19, 3]. Our LightLDA sampler also turns to Metropolis-Hastings, but with new insights into the design of proposal distribution, which is most crucial for high performance. We show that the sampling process can be accelerated even further with a well-designed proposal distribution $q(\cdot)$ to the true LDA posterior $p(\cdot)$.

A well-designed proposal $q(\cdot)$ should speed up the sampling process in two ways: (1) drawing samples from $q(\cdot)$ will be much cheaper than drawing samples from $p(\cdot)$; (2) the Markov chain should mix quickly (i.e. requires only a few steps). What are the trade-offs involved in constructing a good proposal distribution $q(\cdot)$ for $p(\cdot)$? If $q(\cdot)$ is close to $p(\cdot)$, then the constructed Markov chain will mix quickly — however, the cost of sampling from $q(\cdot)$ might end up as expensive as sampling from $p(\cdot)$ itself. On the contrary, if $q(\cdot)$ is very different from $p(\cdot)$, we might be able to sample from it cheaply — but the constructed Markov chain may mix too slowly, and require many steps for convergence.

4.2 Cheap Proposals by Factorization

To design an MH algorithm that is cheap to draw from, yet has high mixing rate, we adopt a *factorized* strategy: instead of a single proposal, we shall construct a set of $\mathcal{O}(1)$ proposals, and alternate between them. To construct these proposals, let us begin from the true conditional probability of token topic indicator z_{di}:

$$p(k) = p(z_{di}=k|\text{rest}) \propto \frac{(n_{kd}^{-di}+\alpha_k)(n_{kw}^{-di}+\beta_w)}{n_k^{-di}+\bar{\beta}}. \quad (4)$$

Observe that it can be decomposed into two factors:

$$q(z_{di}=k|\text{rest}) \propto \underbrace{(n_{kd}+\alpha_k)}_{\text{doc-proposal}} \times \underbrace{\frac{n_{kw}+\beta_w}{n_k+\bar{\beta}}}_{\text{word-proposal}}. \quad (5)$$

Even if we exploit sparsity in both terms, sampling from this conditional probability costs at least $\mathcal{O}(\min(K_d, K_w))$. To do better, we utilize the following observation: the first term is document-dependent but word-independent, while the second term is doc-independent but word-dependent. Furthermore, it is intuitive to see that the most probable topics are those with high probability mass from both the doc-dependent term and the word-dependent term; hence, either term alone can serve as a good proposal q — because if p has high probability mass on topic k, then either term will also have high probability mass on k (though the converse

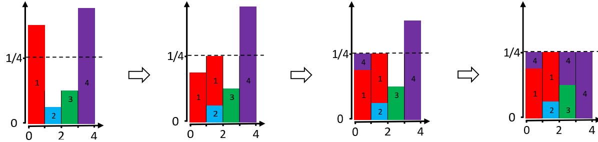


Figure 2: An example showing how to build an alias table. This procedure transforms a non-uniform sampling problem into a uniform sampling one. The alias table maintains the mass of each bin and can be re-used once constructed. More details about the alias method can be found in [14].

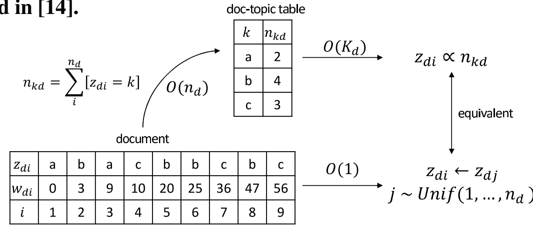


Figure 3: Illustration of how we sample the doc-proposal in $\mathcal{O}(1)$ time, without having to construct an alias table.

is not true). Just as importantly, both factors keep relatively stable during the sampling process[2], so that the alias method [14] (also used in AliasLDA [11]) can be applied to both factors to reduce the sampling cost from either proposal (where the cost of constructing the alias table is getting amortized). We now discuss the proposals individually.

Word-Proposal for Metropolis-Hastings. Define p_w as the word-proposal distribution

$$p_w(k) \propto \frac{n_{kw} + \beta_w}{n_k + \bar{\beta}}. \tag{6}$$

The acceptance probability of state transition $s \to t$ is

$$\min\{1, \frac{p(t)p_w(s)}{p(s)p_w(t)}\}. \tag{7}$$

Let $\pi_w := \frac{p(t)p_w(s)}{p(s)p_w(t)}$, we can show that

$$\pi_w = \frac{(n_{td}^{-di}+\alpha_t)(n_{tw}^{-di}+\beta_w)(n_s^{-di}+\bar{\beta})(n_{sw}+\beta_w)(n_t+\bar{\beta})}{(n_{sd}^{-di}+\alpha_s)(n_{sw}^{-di}+\beta_w)(n_t^{-di}+\bar{\beta})(n_{tw}+\beta_w)(n_s+\bar{\beta})}. \tag{8}$$

Once $t \sim p_w(t)$ is sampled, the acceptance probability can be computed in $\mathcal{O}(1)$ time, as long as we keep track of all sufficient statistics $n.$ during sampling. Intuitively, π_w is high (relative to topic s) whenever the proposed topic t is either (1) popular within document d, or (2) popular for the word w. Since the word-proposal tends to propose topics t which are popular for word w, using the word-proposal will help explore the state space of $p(k)$. We do not need to worry about the situation that the proposed state t is popular for word w but not for the true conditional probability $p(k)$, since in this case, t will be very likely to be rejected due to the low acceptance probability π_w used in the MH process. To sample from p_w in $\mathcal{O}(1)$, we use alias table similar to [11]. As illustrated by Figure 2, the basic idea of the alias approach is to transform a non-uniform distribution into a uniform distribution (i.e., alias table). Since the alias table will be re-used in MH sampling, the transformation cost gets amortized to $\mathcal{O}(1)$ [3].

Although the alias approach has low $\mathcal{O}(1)$ amortized time complexity, its space complexity is still very high, because the alias table for each word's proposal distribution stores $2K$ values: the splitting point of each bin and the alias value above that splitting point; this becomes prohibitive if we need to store a lot of words' alias tables. Our insight here is that the alias table can be sparsified; specifically, we begin by decomposing $p_w = \frac{n_{kw}}{n_k+\bar{\beta}} + \frac{\beta_w}{n_k+\bar{\beta}}$. We then draw one of the two terms, with probability given by their masses (this is known as a mixture approach). If we draw the first term, we use a pre-constructed alias table (created from n_{kw}, specific to word w) to pick a topic, which is sparse. If we draw the second term, we also use a pre-constructed alias table (created from β_w, common to all words w and thus amortized over all V words) to pick a topic, which is dense. In this way, we reduce both the time and space complexity of building word w's alias table to $\mathcal{O}(K_w)$ (the number of topics word w participates in).

Doc-Proposal for Metropolis Hastings. Define p_d as the doc-proposal distribution

$$p_d(k) \propto n_{kd} + \alpha_k. \tag{9}$$

The acceptance probability of state transition $s \to t$ is

$$\min\{1, \frac{p(t)p_d(s)}{p(s)p_d(t)}\}. \tag{10}$$

Let $\pi_d := \frac{p(t)p_d(s)}{p(s)p_d(t)}$, we can show that

$$\pi_d = \frac{(n_{td}^{-di}+\alpha_t)(n_{tw}^{-di}+\beta_w)(n_s^{-di}+\bar{\beta})(n_{sd}+\alpha_s)}{(n_{sd}^{-di}+\alpha_s)(n_{sw}^{-di}+\beta_w)(n_t^{-di}+\bar{\beta})(n_{td}+\alpha_t)}. \tag{11}$$

As with the word-proposal, we see that the doc-proposal accepts whenever topic t is popular (relative to topic s) within document d, or popular for word w. Again, we do not need to worry about the case that the proposed state t is popular within the document but not for the true conditional probability $p(k)$, since in this case, t will be very likely to be rejected due to the low acceptance probability π_d used in the MH process. We decompose $p_d(k) \propto \frac{n_{kd}}{n_d+\bar{\alpha}} + \frac{\alpha_k}{n_d+\bar{\alpha}}$ just like the word-proposal, except that when we pick the first term, we do not even need to explicitly build an alias table — this is because the document token topic indicators z_{di} serve as the alias table by themselves. Specifically, the first term n_{kd} counts the number of times topic k occurs in document d, in other words

$$n_{kd} = \sum_{i=1}^{n_d} [z_{di} = k], \tag{12}$$

where $[\cdot]$ is the indicator function. This implies that the array z_{di} is an alias table for the unnormalized probability distribution n_{kd}, and therefore we can sample from n_{kd} by simply drawing an integer j uniformly from $\{1, 2, \ldots, n_d\}$, and setting $z_{di} = z_{dj}$. Figure 3 uses a toy example to illustrate this procedure. Hence, we conclude that the doc-proposal can be sampled in $\mathcal{O}(1)$ non-amortized time (because we do not need to construct an alias table)[4].

4.3 Combining Proposals to Improve Mixing

While either the doc- or word-proposal alone can be used as an as efficient MH algorithm for LDA, in practice many MH-steps (repeatedly sampling each token) are required to produce proper mixing. With a small number of MH-steps, using the word-proposal alone encourages sparsity in word-topic distribution (i.e. each word belongs to few topics) but causes low sparsity in document-topic distributions (i.e. each document contains many topics). Conversely, using the doc-proposal alone with few MH-steps leads to sparsity in document-topic distribution but non-sparse word-topic dis-

[2]This is crucial because alias tables must be re-used to amortize computational complexity, which in turn requires the target distribution to be stable enough. This also explains why the decomposition in AliasLDA can not lead to an $\mathcal{O}(1)$ algorithm, that is, the first term in Eq. 3 involves the product of n_{kd} and n_{kw}, and is not stable enough to use alias approach.

[3]This strategy keeps the Metropolis-Hastings proposal p_w fixed over multiple documents, rather than changing it after every token. This is well-justified, since Metropolis-Hastings allows any proposal (up to some conditions) provided the acceptance probability can be correctly computed. We have already argued in Section 4.2 that the acceptance probability can be computed in $\mathcal{O}(1)$ time by simply keeping track of a few sufficient statistics $n.$.

[4]Unlike the word-proposal, p_d changes after every token to reflect the current state of z_{di}. Again, this is fine under Metropolis-Hastings theory.

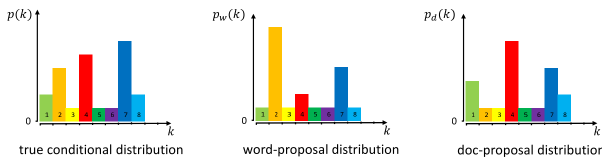


Figure 4: An example that explains why cycling two proposals helps mixing rate. The 2nd bin is a mode of $p(k)$, and $p_d(k)$ is obviously not a good proposal for this mode — but $p_w(k)$ is good at exploring it.

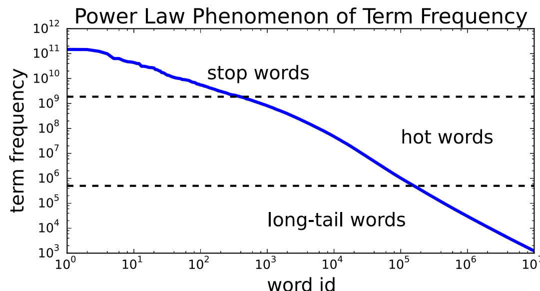


Figure 5: Word frequencies in topic modeling follow a power-law phenomenon (log-log plot). The great difference in the frequency of hot words versus long-tailed words makes selecting the right data structure difficult, as discussed in the text. This plot is obtained from 15 billion web pages, with over than 3000 billion tokens.

tributions. Therefore, while either proposal can sample tokens very quickly, they need many MH-steps to mix well.

The key to fast Metropolis-Hastings mixing is a proposal distribution that can quickly explore the state space, and reach all states with high probability (i.e., the modes). The word-proposal $p_w(k)$ is good at proposing only its own modes (resulting in concentration of words in a few topics), and likewise for the doc-proposal $p_d(k)$ (resulting in concentration of docs onto a few topics). As Figure 4 shows, with the word-proposal or doc-proposal alone, some modes will never be explored quickly.

How can we achieve a better mixing rate while still maintaining high sampling efficiency? If we look at $p(k) \propto p_w(k) \times p_d(k)$, we see that for $p(k)$ to be high (i.e. a mode), we need either $p_w(k)$ or $p_d(k)$ to be sufficiently large — but not necessarily both at the same time. Hence, our solution is to combine the doc-proposal and word-proposal into a "cycle proposal"

$$p_c(k) \propto p_d(k)p_w(k), \tag{13}$$

where we construct an MH sequence for each token by alternating between doc- and word-proposal. The results of [19] show that such cycle proposals are theoretically guaranteed to converge. By combining the two proposals in this manner, all modes in $p(k)$ will be proposed, with sufficiently high probability, by at least one of the proposals. Another potential benefit of cycling different proposals is that it helps to reduce the auto-correlation among sampled states, thus exploring the state space more quickly.

5. HYBRID DATA STRUCTURES FOR POWER-LAW WORDS

Even with carefully designed data parallelization and model scheduling, RAM capacity remains a critical obstacle when scaling LDA to very large number of topics. The LDA model, or *word-topic table* n_{kw}, is a $V \times K$ matrix, and a naive dense representation would require prohibitive amounts of memory. For example, for $V = K = 1$ million used in this paper's experiments, the model would be 4 terabytes in size assuming 32-bit integer entries. Even with reasonably well-equipped machines each with 128 gigabytes of RAM, just storing the matrix in memory would require 32 machines. In practice, the actual usage is often much higher due to other system overheads (e.g. cache, alias tables, buffers, parameter server).

A common solution is to turn to sparse data structures such as hash maps. The rationale behind sparse storage is that document words follow a power-law distribution (Figure 5). There are two implications: (1) after removing stop words, the term frequency of almost all meaningful words will not exceed the upper range of a 32-bit integer (2,147,483,647); this was measured on a web-scale corpus with 15 billion webpages and over 3000 billion tokens, and only 300 words' term frequencies exceed the 32-bit limit. For this reason, we choose to use 32-bit integers rather than 64-bit ones. (2) Even with several billion documents, the majority of words occur fewer than K times (where K may be up to 1 million in our experiments). This means that most rows $n_{k,\cdot}$ in the *word-topic table* are extremely sparse, so a sparse row representation (hash maps) will significantly reduce the memory footprint.

However, compared to dense arrays, sparse data structures exhibit poor random access performance, which hurts MCMC algorithms like SparseLDA, AliasLDA and LightLDA because they all rely heavily on random memory references. In our experiments, using pure hash maps results in a several-fold performance drop compared to dense arrays. How can we enjoy low memory usage whilst maintaining high sampling throughput? Our solution is a hybrid data structure, in which *word-topic table* rows corresponding to frequent words are stored as dense arrays, while uncommon, long-tail words are stored as open-addressing/quadratic-probing hash tables. In our web-scale corpus with several billion documents, we found that the top 10% frequent words in the vocabulary cover almost 95% of all tokens in the corpus, while the remaining 90% of vocabulary words are long-tail words that cover only 5% of the tokens. This implies that (1) most accesses to the hybrid *word-topic table* go to dense arrays, which keeps throughput high; (2) most rows of the *word-topic table* are still sparse hash tables[5], which keeps memory usage reasonably low. In our $V = K = 1$ million experiments, the hybrid *word-topic table* used 0.7TB, down from 4TB if we had used purely dense arrays. When this table is distributed across 24 machines, only 30GB per machine is required, freeing up valuable memory for other system components.

6. SYSTEM IMPLEMENTATION

Distributed implementations are clearly desirable for web-scale data: they reduce training time to realistic levels, and most practitioners have access to at least a small distributed cluster. However, existing distributed LDA implementations have only been shown to work at much smaller problem scales (particularly model size), or suggest the use of extremely large computer clusters (sometimes numbering in the thousands of machines) to finish training in acceptable time. What are the challenges involved in solving big LDA problems on just tens of machines? If we want to train on a corpus with billions of documents (each with at least hundreds of tokens) occupying terabytes of space, then on the data side, simply copying the data from disk to memory will take tens of hours, while transferring the data over the network also takes a similar amount of time. On the model side, storing 1 trillion parameters (1 million words by 1 million topics) can take up to terabytes of memory — necessitating distributed storage, which in turn requires inter-machine parameter synchronization, and thus high network communication cost. In light of these considerations, our goal is to design an architecture for LightLDA that reduces these data transmission and

[5]In order to further improve the throughput of the long tail words, we set the capacity of each hash table to at least two times the term frequency of a long-tail word. This guarantees a load factor that is ≤ 0.5, thus keeping random access performance high.

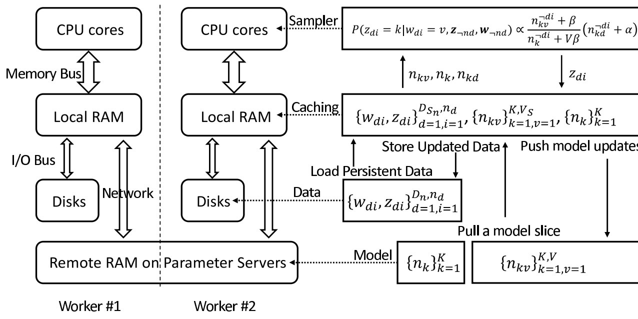


Figure 6: **System architecture, data/model placement and logical flow.**

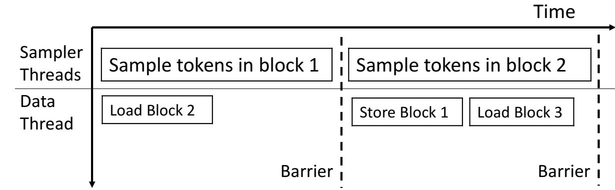
parameter communication costs as much as possible, thus making execution on small clusters realistic.

System Overview. We build LightLDA on top of an open-source framework for distributed large-scale ML, Petuum (`www.petuum.org`). We specifically make use of its parameter server [18, 9, 12] for bounded-asynchronous data-parallelism. We first introduce the general parameter server idea, and then describe our substantial enhancements to make big LDA models possible on small clusters.

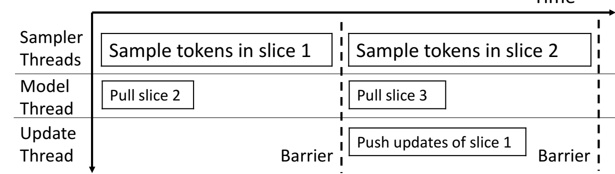
Parameter Server and Data Placement. At the basic level, our parameter server (PS) presents a distributed shared memory interface [9], where programmers can access any parameter from any machine, agnostic to the physical location of the parameter. Essentially, the PS extends the memory hierarchy of a single machine (Figure 6); storage media closer to the CPU cores has lower access latency and higher transmission bandwidth, but has much smaller capacity. In the PS architecture, each machine's RAM is split into two parts: local RAM for client usage and remote RAM for distributed parameter storage (referred to as the "server" part). These hardware limitations, together with the requirements imposed by big topic model, strongly influence the manner in which we run our Metropolis-Hastings algorithm.

We use the PS to store two types of LDA model parameters: the *word-topic table* $\{n_{kv}\}_{k=1,v=1}^{K,V}$, which counts the number of tokens with word v assigned to topic k, and a length-K "summary row" $\{n_k\}_{k=1}^{K}$ which counts the total number of tokens assigned to topic k (regardless of word). 32-bit integers are used for the *word-topic table* (using a combination of dense arrays and sparse hash maps; see Section 5), and a 64-bit integer array for the summary row. We observe that as the LDA sampler progresses, the *word-topic table* becomes increasingly sparse, leading to lower network communication costs as time passes. Furthermore Petuum P-S supports a bounded-asynchronous consistency model [9], which reduces inter-iteration parameter synchronization times through a staleness parameter s — for LightLDA, which is already a heavily-pipelined design, we found the optimal value to be $s = 1$.

Given that the input data are much larger than the model (and their volume remains unchanged throughout LDA inference), it is unwise to exchange data over the network. Instead, we shuffle and shard the corpus across the disks of all worker machines, and each worker machine only ever accesses the data in its local disk. In Figure 6, $\{w_{di}, z_{di}\}_{d=1,i=1}^{D_n,n_d}$ indicates a shard of training data in the n-th worker machine, where D_n represents the number of documents in the n-th worker, n_d indicates the number of tokens in document d. Each worker's local memory holds (1) the active working set of data $\{w_{di}, z_{di}\}_{d=1,i=1}^{D_{S_n},n_d}$, and (2) the model $\{n_{kv}\}_{k=1,v=1}^{K,V_S}$ required to sample the current set of tokens (using the Metropolis-Hastings sampler). During sampling, we update the token topic indicators z_{di}, and the *word-topic table*. The token-topic pairs ((w_{di}, z_{di}) are local to the worker machine and incur no network communication,



(a) Data block pipeline



(b) Model slice pipeline

Figure 7: **Pipelines to overlap computation, disk I/O and network.**

while the *word-topic table* is stored in the PS and therefore requires a background thread for efficient communication.

Token and Topic Indicator Storage. In data-parallel execution, each worker machine stores a corpus shard on its local disk. For web-scale corpora, each shard may still be very large — hundreds of gigabytes, if not several terabytes — which prohibits loading the entire shard into memory. Thus, we further split each shard into data blocks, and stream the blocks one at a time into memory (Figure 1 left). Data-structure-wise, we deliberately place tokens w_{di} and their topic indicators z_{di} side-by-side, as a vector of (w_{di}, z_{di}) pairs rather than two separate vectors for tokens and topic indicators (which was done in [1]). We do this to improve data locality and CPU cache efficiency: whenever we access a token w_{di}, we always need to access its topic indicator z_{di}, and the vector-of-pairs design directly improves locality. One drawback to this design is extra disk I/O, from writing the (unchanging) tokens w_{di} to disk every time a data shard gets swapped out. However, disk I/O can always be masked via pipelined reads/writes, done in the background while the sampler is processing the current shard.

We point out that our model-scheduling and disk-swapping (out-of-core) design naturally facilitates fault tolerance: if we perform data swapping to disk via atomic file overwrites, then whenever the system fails, it can simply resume training via warm-start: read the swapped-to-disk data, re-initialize the word-topic table, and carry on. In contrast, for LDA systems like PLDA+ [13] and YahooLDA [1] to have fault recovery, they would require periodic dumps of the data and/or model — but this incurs a nontrivial cost in the big data/model scenarios that we are trying to address.

Tuning the Model Scheduling Scheme. In Section 3, we introduced the high-level idea of model scheduling and its applications to LDA. There are still a number of improvements that can be employed to improve its efficiency. We present the most notable ones:

1. After completing a data block or a model slice, a worker machine's CPU cores need to wait for the next data block/model slice to be loaded from disk/network respectively. We eliminate this latency via pipelining (Figure 7). Specifically, the pipelining is achieved via a so-called double-buffering technique[4], which requires double memory quotas but can successfully overlap the computation and the communication (i.e., keeps the C-PUs always busy). We caution that perfect pipelining requires careful parameter configuration (taking into consideration the throughput of samplers, size of data blocks, size of model s-lices).
2. To prevent data load imbalances across model slices, we generate model slices via sorting the vocabulary by word frequencies, and then shuffling the words. In this manner, each slice

will contain both frequent words and long tail words, improving load balance.

3. To eliminate unnecessary data traversal, when generating data blocks, we sort token-topic pairs (w_{di}, z_{di}) according to w_{di}'s position in the shuffled vocabulary, ensuring that all tokens belonging to the same model slice are actually contiguous in the data block (see Figure 1). This sorting only needs to be performed once, and is very fast on data processing platforms like Hadoop (compared to the LDA sampling time). We argue that this is more efficient than the "word bundle" approach in PLDA+ [13], which uses an inverted index to avoid data traversal, but at the cost of doubling data memory requirements.
4. We use a bounded asynchronous data parallel scheme [9] to remove the network waiting time occurring at the boundary of adjacent iterations. Note that, to pre-fetch the first slice of model for the next data block, we do not need to wait for the completion of the sampling on the current data block with the last slice of the model. This is exactly what we call Stale Synchronous Parallel (SSP) programming model.
5. We pre-allocate all the required memory blocks for holding data, model and local updates respectively. In other words, there is no any dynamic memory allocations during the whole training process. This helps remove the potential contention at the system heap lock. According to the specified memory quotas for either data or model, the system will automatically adjust the width of model slice for scheduling and streaming.

Multi-thread Efficiency. Our sampler is embarrassingly parallel within a single worker machine. This is achieved by splitting the in-memory data block into disjoint partitions (to be sampled by individual threads), and sharing the immutable in-memory model slice amongst the sampling threads. Furthermore, we make the shared model slice immutable, and all the updates to the model are firstly locally aggregated and then sent to be globally aggregated at the parameter server. By keeping the model slice immutable, we eliminate the race conditions of concurrent writing to a shared memory region, thus achieving near-linear intra-node scalability. While delaying model updates theoretically slows down the model convergence rate, in practice, it eliminates concurrency issues and thus increases sampler throughput, easily outweighing the slower convergence rate.

Modern server-grade machines contain several CPU sockets (each CPU houses many physical cores) which are connected to separate memory banks. While these banks can be addressed by all CPUs, memory latencies are much longer when accessing remote banks attached to another socket — in other words, Non-Uniform Memory Access (NUMA). In our experiments, we have found NUMA effects to be fairly significant, and we partially address them through tuning sampling parameters such as the number of Metropolis-Hastings steps (which influences CPU cache hit rates, and mitigates NUMA effects). That said, we believe proper NUMA-aware programming is a better long-term solution to this problem. Finally, we note that setting core affinities for each thread and enabling hardware hyper-threading on Intel processors can be beneficial; we observed a 30% performance gain when employing both.

7. EXPERIMENTAL RESULTS

We demonstrate that LightLDA is able to train much larger LDA models on similar or larger data sizes than previous LDA implementations, using much fewer machines — due to the carefully designed data parallelization and model scheduling, and especially the new Metropolis-Hastings sampler that is nearly an order of magnitude faster than SparseLDA and AliasLDA. We use several datasets (Table 7), notably a Bing "web chunk" dataset with 1.2 billion webpages (about 200 billion tokens in total). Our experiments show that (1) the distributed implementation of LightLDA has near-linear scalability in the number of cores and machines; (2) the LightLDA Metropolis-Hastings sampler converges significantly faster than the state-of-the-art SparseLDA and AliasLDA samplers (measured in a single-threaded setting); (3) most importantly, LightLDA enables very large data and model sizes to be trained on as few as 8 machines. Because LightLDA is significantly bigger in model size than previous topic models, and employs a new form of MH algorithm, a comparison on raw throughput (tokens or docs processed per second) that is not calibrated against the model size (as used in previous literature) is less meaningful. In this paper, we focus on absolute speed of convergence (measured by wall clock time to convergence) given the same machine setup, as a universal metric.

DATASET	V	L	D	L/V	L/D
NYTIMES	101636	99542125	299752	979	332
PUBMED	141043	737869083	8200000	5231	90
BING WEBC	1000000	200B	1.2B	200000	167

Table 1: Experimental datasets and their statistics. V denotes vocabulary size, L denotes the number of training tokens, D denotes the number of documents, L/V indicates the average number of occurrences of a word, L/D indicates the average length of a document. For the Bing web chunk data, 200B denotes 200 billion.

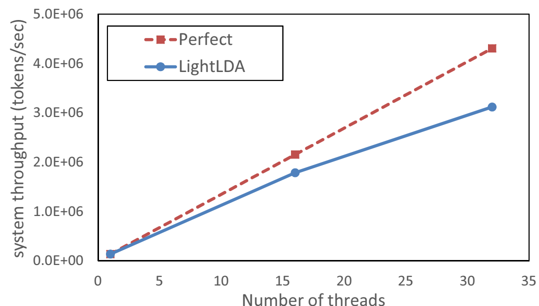


Figure 8: Intra-node scalability of LightLDA, using 1, 16, 32 threads on a single machine (no network communication costs).

7.1 Scalability of Distributed LightLDA

In these experiments, we establish that the LightLDA implementation scales almost linearly in computational resources. We begin with intra-node, single-machine multi-threaded scalability: to do this, we restrict the web chunk dataset to the 50,000 most frequent words, so that the model can be held in the RAM of a single machine. We then run LightLDA with 1, 16 and 32 threads (on a machine with at least 32 logical cores), training a model with 1 million topics on the web chunk dataset. To fully utilize the memory bandwidth, we set the number of Metropolis-Hastings steps as 16 in all the scalability experiments. We record the number of tokens sampled by the algorithm over 2 iterations, and plot it in Figure 8. The figure shows that LightLDA exhibits nearly linear scaling inside a single machine.

Next, we demonstrate that LightLDA scales in the distributed setting, with multiple machines. Using the same web chunk dataset with $V = 50,000$ vocabulary words and $K = 1$ million topics, we run LightLDA on 1, 8 and 24 machines (using 20 physical cores and 256GB RAM per machine[6]), and set the Petuum parameter server to use a staleness value of 1. A positive staleness value lets LightL-

[6]Such high-memory configurations can be readily purchased from cloud compute providers, but are not obligatory for LightLDA. The model-scheduling slices can be made thinner for small-memory machines, and the system will still work.

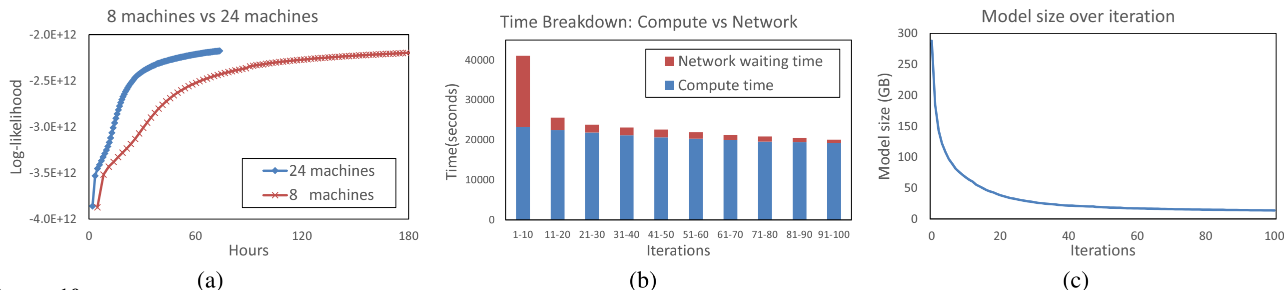


Figure 10: (a) Log-likelihood over running time with 8 and 24 machines, on Bing web chunk dataset, using $V = K = 1$ million (1 trillion parameters). (b) Compute time v.s. Network waiting time, as a function of iteration number. Observe that communication costs are significantly higher during the first 10 iterations, when the model is still dense. (c) Model si

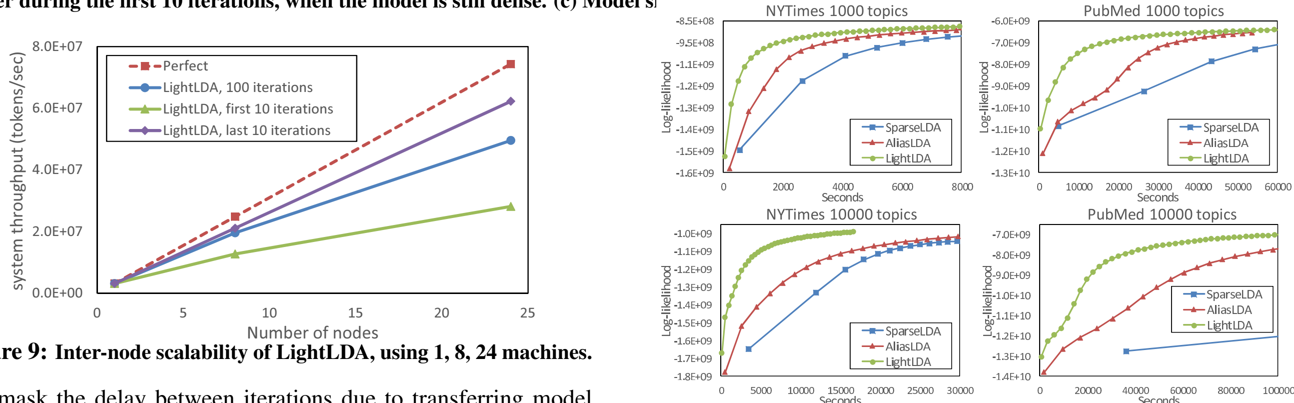


Figure 9: Inter-node scalability of LightLDA, using 1, 8, 24 machines.

Figure 11: Log-likelihood versus running time for SparseLDA, AliasLDA, LightLDA.

DA mask the delay between iterations due to transferring model parameters over the network, yielding higher throughput at almost no cost to quality. This time, we record the number of tokens sampled over 100 iterations, and compute the average token throughput over the first 10 iterations, the last 10 iterations, and all 100 iterations. These results are plotted in Figure 9, and we note that scalability is poor in the first 10 iterations, but close to perfect in the last 10 iterations. This is because during the initial iterations, the LDA model is still very dense (words and documents are assigned to many topics, resulting in large model size; see Figure 10(c)), which makes the system incur high network communication costs (Figure 10(b)) — enough to saturate our cluster's 1 Gbps Ethernet. In this situation, the pipelining (explained in Figure 7) does not mask the extra communication time, resulting in poor performance. However, after the first 10 iterations or so, the model becomes sparse enough for pipelining to mask the communication time, leading to near-perfect inter-node scalability (as shown in Figure 10(c)). To eliminate this initial communication bottleneck, we can initialize the model more sparsely, or we simply upgrade to 10 Gbps Ethernet (frequently seen in industrial platforms and cloud computing services) or better.

Finally, we demonstrate that LightLDA is able to handle very large model sizes: we restore the full $V = 1$ million vocabulary of the web chunk data and $K = 1$ million topics, yielding a total of one trillion model parameters on 200 billion tokens. We then run LightLDA on this large dataset and model using 8 and 24 machines, and plot the log-likelihood curves in (Figure 10(a)). Convergence is observed within 2 days on 24 machines (or 5 days on 8 machines), and it is in this sense that we claim big topic models are now possible on modest computer clusters. It is noteworthy that the system works well with no obvious performance drop on 8 machines even when the vocabulary size is extended from one million to 20 million (i.e., 20 trillion parameters in the model). This is due to the fact that all the extended words will be long-tail ones. With the help of hybrid data structure specifically designed for power-law words, increasing the vocabulary size does not introduce obvious memory footprint and performance load.

One might ask if overfitting happens on such large models, given that the number of parameters (1 trillion) is larger than the number of tokens (200 billion). We point out that (1) there is evidence to show that large LDA models can improve ad prediction tasks [21], and (2) the converged LDA model is sparse, with far fewer than 200 billion nonzero elements. As evidence, we monitored the number of non-zero entries in *word-topic table* during the whole training process, and observed that after 100 iterations, the model had only 2 billion non-zero entries (which is 1% of 200 billion tokens).

7.2 LightLDA Algorithm versus Baselines

We want to establish that our Metropolis-Hastings algorithm converges faster than existing samplers (SparseLDA and AliasLDA) to a high quality model. Using one computational thread, we ran LightLDA, SparseLDA and AliasLDA on two smaller datasets (NYTimes, PubMed) using $K = 1,000$ or 10,000 topics, and plotted the log-likelihood versus running time in Figure 11. We set the number of Metropolis-Hastings step to 2, for both AliasLDA and LightLDA. From the results, we make the following observations: AliasLDA consistently converges faster than SparseLDA (as reported in [11]), while LightLDA is around 3 to 5 times as fast as AliasLDA. To better understand the performance differences, we also plot the time taken by the first 100 iterations of each algorithm in Figure 12. In general, LightLDA has a consistently low iteration time and it does not suffer from slow initial iterations. AliasLDA is significantly faster than SparseLDA on datasets with short documents (PubMed[7]), but is only marginally faster on longer documents (NYTimes). This is reasonable since the per-token complexity of AliasLDA depends on the document length.

Although LightLDA is certainly faster per-iteration than SparseLDA and AliasLDA, to obtain a complete picture of convergence, we must plot the log-likelihood versus each iteration (Figure 14). We observe that SparseLDA makes the best progress per itera-

[7]Note that we use the whole PubMed data set in this experiment, whereas the AliasLDA paper [11] only considered 1% of the total PubMed data.

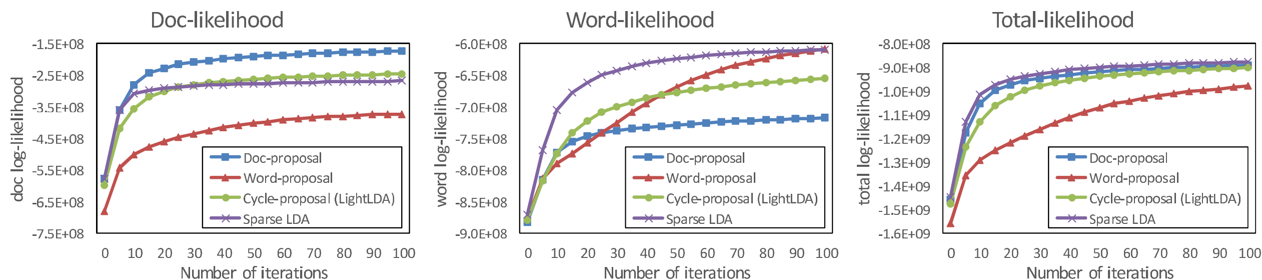


Figure 13: Performance of different LightLDA Metropolis-Hastings proposals, on the NYTimes data set with $K = 1000$ topics.

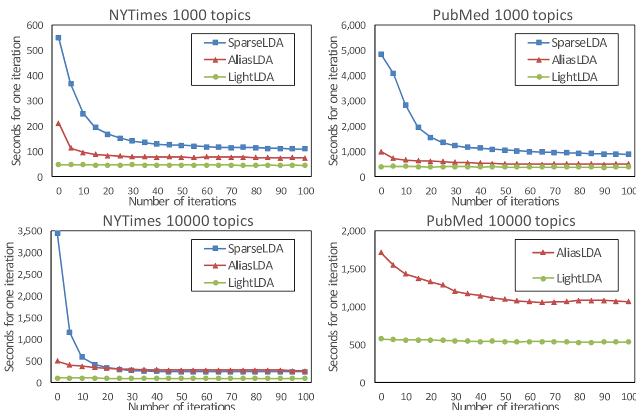


Figure 12: Running time for each of the first 100 iterations of SparseLDA, AliasLDA, LightLDA. The curve for SparseLDA was omitted from the $K = 10,000$-topic PubMed experiment, as it was too slow to show up in the plotted range.

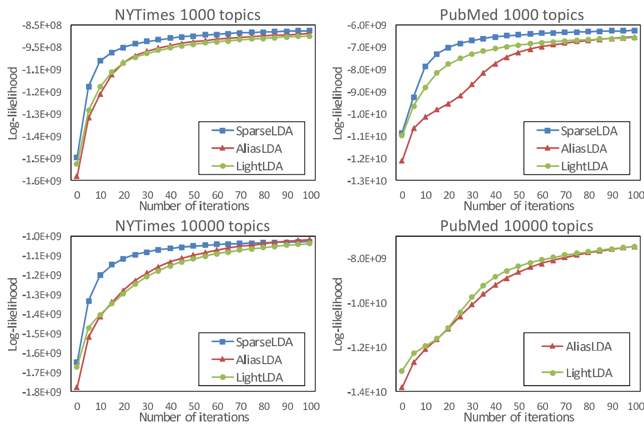


Figure 14: Log-likelihood versus iteration number for SparseLDA, AliasLDA, LightLDA.

tion (because it uses the original conditional Gibbs probability), while LightLDA is usually close behind (because it uses simple Metropolis-Hastings proposals). AliasLDA (another Metropolis-Hastings algorithm) is either comparable to LightLDA (on NYTimes) or strictly worse (on Pubmed). Because the per-iteration time taken for LightLDA and AliasLDA algorithm is very short (Figure 12), in terms of convergence to the same loss level *per time*, LightLDA and AliasLDA are significantly faster than SparseLDA (with LightLDA being the faster of the two, see Figure 11). In summary, LightLDA can achieve the same topic model quality in much less time than the other two algorithms.

Finally, we argue that the "cycle proposal" used in LightLDA, which alternates between the doc-proposal and word-proposal, would be highly effective at exploring the model space, and thus a better option than either proposal individually. To demonstrate this, in Figure 13, we plot the performance of the full LightLDA cycle proposal versus the doc- and word-proposals alone, as well as SparseLDA (which represents the gold standard for quality, being a non-Metropolis-Hastings Gibbs sampler). The plots are subdivided into (1) full likelihood, (2) document likelihood, and (3) word likelihood ([18] explains this decomposition in more detail). Intuitively, a high doc-likelihood shows that the learned topics compactly represent all documents, while a high word-likelihood indicates that the topics compactly represent the vocabulary. Although the doc- and word-proposals appear to do well on total likelihood, in actual fact, the word-proposal fails to maximize the doc-likelihood, while the doc-proposal fails to maximize the word-likelihood. In contrast, the full LightLDA cycle proposal represents both documents and vocabulary compactly, and is nearly as good in quality as SparseLDA. We note that, by trading-off between the word-proposal and doc-proposal, LightLDA allows explicit control of the word-topic distribution sparsity versus the doc-topic distribution sparsity. For real-world applications that require one type of sparsity or the other, this flexibility is highly desirable.

8. CONCLUSIONS

We have implemented a distributed LDA sampler, LightLDA, that enables very large data sizes and models to be processed on a small computer cluster. LightLDA features significantly improved sampling throughput and convergence speed via a (surprisingly) fast $\mathcal{O}(1)$ Metropolis-Hastings algorithm, and allows even small clusters to tackle very large data and model sizes thanks to the carefully designed model scheduling and data parallelism architecture, implemented on the Petuum framework. A hybrid data structure is used to simultaneously maintain good performance and memory efficiency, providing a balanced trade-off. On a future note, we believe the *factorization-and-combination* techniques for constructing Metropolis-Hastings proposals can be successfully applied to the inference of other graphical models[8], alongside the model scheduling scheme as a new enhancement over existing model partitioning strategies used in model-parallelism. It is our hope that more ML applications can be run at big data and model scales on small, widely-available clusters, and we hope that this work will inspire current and future development of large-scale ML systems.

9. ACKNOWLEDGMENTS

The authors thank the reviewers for their comments that helped improve the manuscript. This work is supported in part by DARPA FA87501220324, and NSF IIS1447676 grants to Eric P. Xing.

[8]Given that all probabilistic graphical models have an equivalent description as a factor graph, the factorization and cycling tricks can be used to speedup inference on general graphs when: (1) the factored proposal distribution is stable enough, so that it makes sense to apply the alias approach, and (2) the number of discrete states is so large that sampling from a Multinomial distribution becomes a computation bottleneck.

10. REFERENCES

[1] A. Ahmed, M. Aly, J. Gonzalez, S. Narayanamurthy, and A. J. Smola. Scalable inference in latent variable models. In *WSDM*, pages 123–132, 2012.

[2] E. Airoldi, D. Blei, S. Fienberg, and E. Xing. Mixed membership stochastic blockmodels. *J. Mach. Learn. Res.*, 9:1981–2014, 2008.

[3] C. Andrieu, N. D. Freitas, A. Doucet, and M. I. Jordan. An introduction to MCMC for machine learning. *Machine learning*, 50(1):5–43, 2003.

[4] M. Bauer, H. Cook, and B. Khailany. Cudadma: Optimizing gpu memory bandwidth via warp specialization. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 12:1–12:11, New York, NY, USA, 2011. ACM.

[5] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *J. Mach. Learn. Res.*, 3(30):993–1022, 2003.

[6] W. Dai, A. Kumar, J. Wei, Q. Ho, G. Gibson, and E. P. Xing. High-performance distributed ml at scale through parameter server consistency models. In *AAAI*, 2015.

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

[8] W. K. Hastings. Monte Carlo sampling methods using Markov chains and their applications. *Biometrika*, 57(1):97–109, 1970.

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

[10] S. Lee, J. K. Kim, X. Zheng, Q. Ho, G. A. Gibson, and E. P. Xing. On model parallelization and scheduling strategies for distributed machine learning. In *NIPS*, 2014.

[11] A. Li, A. Ahmed, S. Ravi, and A. J. Smola. Reducing the sampling complexity of topic models. In *KDD*, 2014.

[12] M. Li, D. G. Andersen, J. W. Park, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *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. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 2(3):26, 2011.

[14] G. Marsaglia, W. W. Tsang, and J. Wang. Fast generation of discrete random variables. *Journal of Statistical Software*, 11(3):1–11, 2004.

[15] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equations of state calculations by fast computing machines. *J. Chem. Phys.*, 21:1087–1091, 1953.

[16] D. Newman, A. Asuncion, P. Smyth, and M. Welling. Distributed algorithms for topic models. *The Journal of Machine Learning Research*, 10:1801–1828, 2009.

[17] S. Shringarpure and E. P. Xing. mstruct: inference of population structure in light of both genetic admixing and allele mutations. *Genetics*, 182(2):575–593, 2009.

[18] A. Smola and S. Narayanamurthy. An architecture for parallel topic models. *Proc. VLDB Endow.*, 3(1-2), 2010.

[19] L. Tierney. Markov chains for exploring posterior distributions. *Annals of Statistics*, 22:1701–1762, 1994.

[20] A. J. Walker. An efficient method for generating discrete random variables with general distributions. *ACM Trans. Math. Softw.*, 3(3):253–256, 1977.

[21] Y. Wang, X. Zhao, Z. Sun, H. Yan, L. Wang, Z. Jin, L. Wang, Y. Gao, J. Zeng, Q. Yang, et al. Peacock: Learning long-tail topic features for industrial applications. *arXiv preprint arXiv:1405.4402*, 2014.

[22] L. Yao, D. Mimno, and A. McCallum. Efficient methods for topic model inference on streaming document collections. In *KDD*, 2009.

[23] J. Yin, Q. Ho, and E. P. Xing. A scalable approach to probabilistic latent space inference of large-scale networks. *NIPS*, 2013.

[24] J. Zhu, A. Ahmed, and E. P. Xing. Medlda: maximum margin supervised topic models for regression and classification. In *ICML*, pages 1257–1264, 2009.