Towards Topic Modeling for Big Data

Yi Wang[1], Xuemin Zhao[1], Zhenlong Sun[1], Hao Yan[1], Lifeng Wang[1], Zhihui Jin[1]
Liubin Wang[1], Yang Gao[2], Jia Zeng[2,3], Qiang Yang[3] and Ching Law[1]
[1]Tencent, Peking 100080, China
[2]School of Computer Science and Technology, Soochow University, Suzhou 215006, China
[3]Huawei Noah's Ark Lab, Hong Kong
j.zeng@ieee.org

ABSTRACT

Latent Dirichlet allocation (LDA) is a popular topic modeling technique in academia but less so in industry, especially in large-scale applications involving search engines and online advertisement systems. A main underlying reason is that the topic models used have been too small in scale to be useful; for example, some of the largest LDA models [25, 19, 4, 24, 17, 31] reported in literature have up to 10^3 topics, which cover difficultly the long-tail semantic word sets. In this paper, we show that the number of topics is a key factor that can significantly boost the utility of topic-modeling system. In particular, we show that a "big" LDA model with at least 10^5 topics inferred from 10^9 search queries can achieve a significant improvement on industrial search engine and online advertising system, both of which serving hundreds of millions of users. We develop a novel distributed system called Peacock to learn big LDA models from big data. The main features on Peacock include hierarchical parallel architecture, real-time prediction, and topic de-duplication. We empirically demonstrate that the Peacock system is capable of providing significant benefits via highly scalable LDA topic models for several industrial applications.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Database Management**]: Database Applications—*Data Mining*

General Terms

Algorithms, Experimentation, Performance

Keywords

Latent Dirichlet allocation, big topic models, big data, topic feature, search engine, online advertising system

1. INTRODUCTION

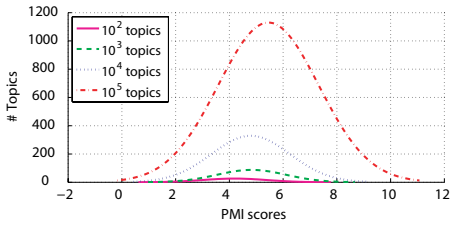

Figure 1: The topic PMI histograms of LDA models with various numbers of topics.

In academia, latent Dirichlet allocation (LDA) [6] is a popular topic modeling technique, which is an unsupervised learning algorithm to infer semantic word sets called topics. However, very few successes of LDA have been reported in industry. The major reason is that the largest LDA models reported in literature [25, 19, 4, 24, 17, 31] have up to 10^3 topics, which cannot cover completely the long-tail semantic word sets in big data. Industrial applications like search engine and online advertising require the capability of learning many semantic word sets (or topics) that cover a large part of human knowledge, in particular, the long-tail part. As reported by Linguistic Data Consortium, there are millions of vocabulary words in either English, Chinese, Spanish or Arabic [14]. Taking polysemy and synonyms under consideration, a rough estimate of the number of word senses is close to the same magnitude of vocabulary words, i.e., 10^5 or 10^6 topics for semantics of long-tail word sets.

To the best of our knowledge, the number of topics in applications mentioned above require topics that are around two to three orders of magnitude larger than the current state-of-the-art [22, 2]. The effectiveness of large number of topics is inspired by [16], which proposes a MapReduce-based *frequent itemset mining* algorithm to find long-tail word sets. We notice that there is a word set containing the words "whorf piraha chomsky anthropology linguistics". Using web search, we find that this word set has a clear semantic meaning on the research by Whorf and Chomsky on anthropology and linguistics based on their study of the Piraha language. However, it is a regret that the *frequent itemset mining* cannot interpret new documents out of the training corpus. Fortunately, LDA overcomes this shortcoming and can infer highly interpretable and semantically coherent topics from new documents.

The point-wise mutual information (PMI) can measure the interpretability or semantic coherence of topics [20]. The

higher PMI score corresponds to a better topic quality. Fig. 1 shows the histograms of topic PMI scores of four LDA models with 10^2, 10^3, 10^4 and 10^5 topics, respectively. With the increase of topics from 10^2 to 10^5, these histograms shift towards larger mean PMI score, and more topics have higher PMI scores. This phenomenon suggests that larger LDA models tend to encode more interpretable and semantically coherent topics.

In this paper, we confirm that big LDA models with at least 10^5 topics can achieve a significant improvement in two online applications such as search engine and online advertising system. This result motivates us to pursue scalable topic modeling methods for big data. To achieve this goal, we develop a distributed system called Peacock that can generate a much lager number of topics than before. For example, Peacock can learn at least 10^5 topics from 10^9 search queries. This improvement is nontrivial and raises many new technical challenges. First, how to make the system scalable to process big query data as well as LDA parameters with fault tolerance? Second, how to do real-time topic prediction for new queries and remove duplicate topics to obtain high-quality topics? Finally, how to integrate big LDA models into existing search engine and online advertising system for a better performance? We address these technical issues and summarize our contributions as follows:

- We design a new hierarchical parallel architecture including model parallelism to handle a large number of LDA parameters as well as data parallelism to handle massive training corpus. We also use pipeline and lock-free techniques to reduce communication and synchronization costs. This architecture runs on a computer cluster including thousands of CPU cores, which can learn $\geq 10^5$ topics from $\geq 10^9$ search queries, around two orders of magnitude larger than the current state-of-the-art reported in literature [22, 2].
- When do topic modeling for big data, we solve two new practical problems in real-world applications: real-time prediction and topic de-duplication. A new real-time prediction algorithm called RT-LDA is developed to infer the topic distributions of unseen queries in search engine and online advertising system. We use two methods for topic de-duplication: 1) Learning asymmetric Dirichlet priors [23]; 2) Clustering similar topics by their L_1-similarities.
- We examine the effectiveness of big LDA models in two online applications: search engine and online advertising system. The performance improvements in both systems grow with the increasing number of the learned topics. We observe a significant improvement on search relevance when the number of topics increases from 10^3 to 10^4. The topic features significantly improve the accuracy of ad click-through rate prediction when the number of topics increases from 10^4 to 10^5.

In the next section, we briefly introduce the background and related work. In Section 3, we present the hierarchical parallel architecture, real-time prediction and topic de-duplication in Peacock system. In Section 4, we show that Peacock is more scalable to larger number of topics than the state-of-the-art industrial solution Yahoo!LDA [22, 2]. Section 5 shows two online applications of Peacock: search engine and online advertising system. In Section 6, we make conclusions and envision future work.

2. BACKGROUND AND RELATED WORK

There are four categories of inference algorithms proposed to estimate LDA parameters: variational EM [6], expectation propagation (EP) [18], belief propagation (BP) [29, 28] and Gibbs sampling (GS) [15]. Most distributed LDA systems adopt the GS algorithm [19, 4, 24, 17, 22, 25], because it requires significantly less memory than other algorithms. The GS algorithm maintains three LDA parameter count matrices in memory: a matrix $\Phi_{V\times K}$ in which each element is the number of vocabulary word $v, 1 \leq v \leq V$ assigned to the topic $k, 1 \leq k \leq K$, a matrix $\Theta_{K\times D}$ in which each element is the number of topic k assignments in document $d, 1 \leq d \leq D$. and a count vector $\Psi_K = \sum_{v=1}^{V} \Phi_{V\times K}$, in which each element is the number of topic k assignments in the training corpus. In each iteration, the GS algorithm updates the topic assignment $z_{ivd} = k$ of every observed word token $x_{ivd} = 1$ in the training corpus by drawing a sample from the collapsed posterior distribution,

$$P(z_{ivd} = k \mid \boldsymbol{z}_{\neg ivd}, \boldsymbol{x}_{\neg ivd}, x_{ivd} = 1, \alpha, \beta) \propto \frac{\Phi_{vk}^{\neg id} + \beta}{\Psi_k^{\neg id} + V\beta} \left(\Theta_{kd}^{\neg id} + \alpha\right), \quad (1)$$

where $\neg ivd$ means that the corresponding word token is excluded in the count matrices. After the topic assignment $z_{ivd} = k$ is sampled, the parameter matrices Ψ_k, Φ_{vk} and Θ_{kd} are updated immediately.

Since we aim to learn $K \geq 10^5$ topic from $D \geq 10^9$ search queries, we choose to distribute an accelerated sparse Gibbs sampling inference (SparseLDA) [26], whose time and space complexities are insensitive to the number of topics K. First, SparseLDA stores only the topic assignment vector $\boldsymbol{z}$ rather than the matrix $\Theta_{K\times D}$, where the size of $\boldsymbol{z}$ is equal to the number of word tokens in corpus, which is irrelevant with the number of topics. When computing Eq. (1) for each document, SparseLDA re-organizes the corresponding $\boldsymbol{z}_{ivd}$ into the document-specific vector Θ_{kd} on the fly. Second, if a vocabulary word v has a total of $N \ll K$ occurrences in training corpus, the vth row of the parameter matrix Φ_{vk} only needs to store up to N rather than K values.

Previous distributed LDA systems have explored two main parallel architectures: (1) parallel computing using processors tightly coupled with shared memory [25], and (2) distributed computing using processors loosely connected via network [19, 4, 24, 17, 31]. Yahoo!LDA [22, 2] can be viewed as a hybrid parallel architecture using the shared memory technique over multiple machines based on *memcached*. All reported distributed LDA systems learn up to 10^3 topics, while 10^3 might be far less than the real number of semantics or word senses in human language. On the other hand, LDA is a non-negative matrix factorization method [29], so that many parallel matrix factorization architectures [12, 32] can be used. However, recent parallel matrix factorization architectures [12, 32] are difficult to learn 10^5 topics because they focus on only low-rank approximation to the big sparse matrix. The rank is often very low such as $10^2 \sim 10^3$, where the rank in matrix factorization has the same meaning with the number of topics in LDA.

Unlike previous distributed LDA solutions, Peacock introduces both data and model parallelism for big data ($\geq 10^9$

documents) and big LDA models ($\geq 10^9$ parameters including topic assignment vector and topic-word matrix) within a hierarchical parallel architecture, which uses pipeline and lock-free techniques to reduce both communication and synchronization costs. In addition, Peacock has two new components like real-time prediction and topic de-duplication, which play important roles in real-world applications.

3. THE PEACOCK SYSTEM

Peacock parallelizes the SparseLDA [26] algorithm for big data. The key challenge is to store one large data set $\boldsymbol{x}_{D\times W}$, one large LDA parameter matrix $\Phi_{V\times K}$, and topic assignment vector $\boldsymbol{z}$, when $V \geq 10^5$, $K \geq 10^5$ and $D \geq 10^9$ in industrial applications. For example, the $\Phi_{V\times K}$ matrix alone needs at least tens of gigabytes when learning 10^5 topics, while modern computer clusters are composed of commodity computers [10] with few gigabytes of memory (e.g., 2GB memory). Therefore, we propose a hierarchical parallel architecture to solve this large-scale problem as shown in Fig. 2, which contains two layers of parallelism to handle both big data and big LDA model.

3.1 Hierarchical Parallel Architecture

In the layer 1 in Fig. 2, we partition the parameter $\Phi_{K\times V}$ matrix by columns $1 \leq v \leq V$ into $1 \leq m \leq M$ model shards, $\{\Phi^1_{vk}, \ldots, \Phi^m_{vk}, \ldots, \Phi^M_{vk}\}$. The value of M should make each model shard Φ^m_{vk} small enough to fits in the memory of a *sampling server*. We also partition the word tokens $\boldsymbol{x}_{D\times V}$ and their topic assignments $\boldsymbol{z}$ by rows into M shards, $\{\boldsymbol{x}^1_{idv}, \boldsymbol{z}^1_{idv}, \ldots, \boldsymbol{x}^m_{idv}, \boldsymbol{z}^m_{idv}, \ldots, \boldsymbol{x}^M_{idv}, \boldsymbol{z}^M_{idv}\}$. The value of M should make each data shard and its corresponding $\boldsymbol{z}$ shard small enough to fit in the memory of a *data server*. In parallel GS inference, we further partition each mth data shard $\{\boldsymbol{x}^m_{idv}, \boldsymbol{z}^m_{idv}\}$ into M parts by columns, and send them to M sampling server for parallel computing. As a result, the entire data has been partitioned into $M \times M$ blocks and each data server store M blocks. We refer to each $M \times M$ shard as a *segment*. Fig. 2 shows a segment with $M = 3$.

If the sampling server simultaneously obtains the same column of data segment sent by all data servers, it will read and write the topic assignments of the same vocabulary word with race conditions. For example in Fig. 2 layer 1, if the sampling server 1 receives simultaneously three data blocks $\{1, 3, 2\}$ sent by three data servers, it has a higher likelihood to change the topic assignments of the same vocabulary words, which causes serious read/write locks and I/O delays. To solve this problem, we design a lock-free parallel strategy similar to [25, 12, 32]. Sampling servers process blocks on the main diagonal sent by data servers. Since only Φ^1_{vk} and $\{\boldsymbol{x}^1_{ivd}, \boldsymbol{z}^1_{ivd}\}$ are required for processing the first block, Φ^2_{vk} and $\{\boldsymbol{x}^2_{ivd}, \boldsymbol{z}^2_{ivd}\}$ for the second, and Φ^3_{vk} and $\{\boldsymbol{x}^3_{ivd}, \boldsymbol{z}^3_{ivd}\}$ for the third, these three blocks can be processed simultaneously without conflicts in accessing Φ_{vk} and $\{\boldsymbol{x}_{ivd}, \boldsymbol{z}_{ivd}\}$. It is analogous to processing blocks on the second and the third diagonals. Because the global parameter Ψ_k is needed for all block computation, we store a local copy of Ψ^m_k in each sampling server and synchronize Ψ^m_k by a coordinator server after processing each diagonal of blocks.

In the layer 1 (Fig. 2), the distributed algorithm can be executed by a configuration of the following servers:

1. *Model parallelism*: We use M *sampling servers*, where the mth sampling server maintains the Φ^m_{vk} shard and local copies of Ψ^m_k and $\boldsymbol{\alpha}_k$ in memory. It runs the SparseLDA [26] algorithm to update topic assignments $\boldsymbol{z}_{ivd}$ in training blocks sent from data servers. The SparseLDA algorithm will update the Φ^m_{vk} shard at mth sampling server. At the end of the training process, all sampling servers output their Φ^m_{vk} shards.

2. *Data parallelism*: We use M *data servers*, where each loads a data shard $\boldsymbol{x}^m_{idv}$ and corresponding $\boldsymbol{z}^m_{idv}$ shard in memory. Data servers send data and topic assignments shards to sampling servers, which updates topic assignments and model parameters. After processing each corpus segment, data servers write the updated corpus shards back to disk.

The scalability of layer 1 is limited. Increasing the number of sampling servers M indicates the increasing of vertical partitions of the training corpus $\boldsymbol{x}_{D\times W}$ as well as model parameters $\Phi_{K\times V}$. Since the vocabulary size is often a fixed value, M cannot be very large in practice. Therefore, we need to build multiple layer 1 configurations and use a set of M *aggregation servers* to aggregate the global model parameter Φ_{vk} from different configurations as shown in layer 2 (Fig. 2 shows three layer 1 copies and three aggregation servers to connect corresponding sampling servers.). At the end of each GS iteration in Fig. 3, all sampling servers in all configurations report their model change $\Delta\Phi^m_{vk}$ to aggregation servers [19, 2]. Notice that the mth sampling server in layer 1 configuration reports only to the mth aggregation server in layer 2. After the aggregation from all configurations, the aggregation servers in layer 2 distribute the updated global model $\Phi_{V\times K}$ to all configurations in layer 1. In practice, the aggregation does not have to be done in every iteration. Instead, layer 1 configurations can run independently for several iterations before the model aggregation in layer 2. Recent stochastic analysis [30] shows that this communication scheme does not influence the convergence of the inference algorithm.

In layer 2, we also use a *coordinator* to control the cooperative work of sampling, data and aggregation servers. The coordinator is also responsible for updating and re-distributing the global copy of Ψ_k to sampling servers, and optimizing the asymmetric Dirichlet prior $\boldsymbol{\alpha}_k$. To optimize $\boldsymbol{\alpha}_k$, the coordinator has to maintain a histogram of document lengths, l_d, and collect a statistics, Ω_{kn}, the number of documents in which topic k appears n times [23] from data servers.

All these servers are developed using Google's Go programming language, an compiled concurrent system programming language. The parallel machine learning systems could benefit a lot from the native support of concurrent programming and convenient implementation of remote procedure calls (RPC) provided by Go.

3.1.1 Distributed GS Algorithm

Fig. 3 shows the distributed GS algorithm executed by the coordinator in layer 1, where the dot symbol denotes an RPC call. For example, in line 6, the coordinator calls procedure SetAlpha, which is exposed and executed by a sampling server. The **par-for** in Fig. 3 denotes the concurrent version of the commonly-used control structure **for**. The **par-for** flattens the loop body and executes it in parallel. The **par-for** can be implemented using Go language elements of *channel* and *goroutine*, as shown by the open source project `https://github.com/wangkuiyi/parallel`,

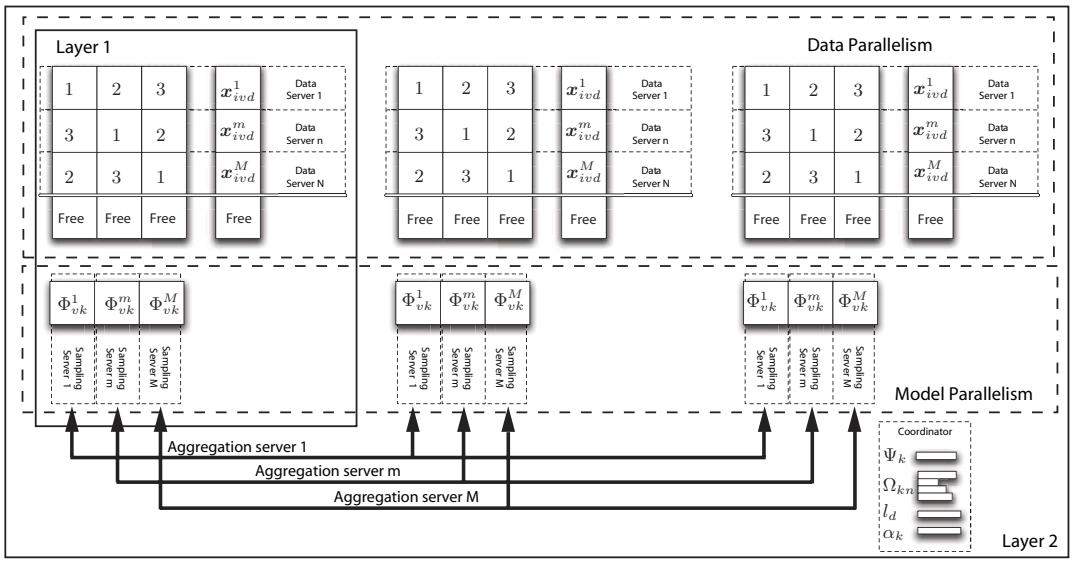

Figure 2: The hierarchical parallel architecture in Peacock. The first layer contains M data servers and M sampling servers for both data and model parallelism. The second layer containts M aggregation servers and one coordinator server for global parameter synchronization and asymmetric prior estimation. This architecture can readily scale up to hundreds of machines having thousands of cores to learn at least 10^5 topics from 10^9 search queries.

```
1: procedure RunGibbsIteration(M, M)
2:   for segment ← 1..⌈M⌉ do
3:     SampleSegment(segment)
4:   α ← OptimizeHyperparams(α, l_d, Ω_kn)
5:   par-for m ← 1..M
6:     sampling-server[m].SetAlpha(α)

7: procedure SampleSegment(segment)
8:   par-for m ← 1..M
9:     data-server[m].LoadShard(segment, m)
10:  for dig ← 1..M do                ▷ dig indices diagonals
11:    par-for m ← 1..M
12:      data-server[(m+dig)%M].WorkWithSampler(m)
13:    par-for m ← 1..M
14:      Ψ_k ← Ψ_k+ sampling-server[m].GetDiffNt
15:    par-for m ← 1..M
16:      sampling-server[m].SetNt(Ψ_k)
17:  par-for m ← 1..M
18:    data-server[m].SaveShard(segment, m)
19:  par-for m ← 1..M
20:    Ω_kn ← Ω_kn+ data-server[m].CountNtn
```

Figure 3: The distributed GS algorithm.

which is used in Peacock. C/C++ programmers can use the **par-for** provided by OpenMP.

In each iteration, the procedure RunGibbsIteration invokes procedure SampleSegment to update topic assignments segment-by-segment, where SampleSegment coordinates the M data servers and the M sampling servers to run the parallel GS algorithm. SampleSegment also collects the matrix, Ω_{kn}, which counts the number of documents in which the topic assignment k occurs for n times.

T	L (KB)	Time (minutes)
200,000	1	48.1
20,000	10	45.3
2,000	100	43.5
200	1,000	43.3
40	5,000	43.4
20	10,000	43.5
10	20,000	44.1
1	200,000	49.8

Table 1: Parameters of the communication pipeline and the corresponding communication time.

This matrix is used by procedure OptimizeHyperparams, which is described in [23], to optimize the asymmetric prior, $\boldsymbol{\alpha}$, at the end of each GS iteration. In addition to Ω_{kn}, OptimizeHyperparams also requires l_d, the vector recording the document lengths, which is counted in the first iteration and saved in coordinator for later use.

3.1.2 Pipeline for Efficient Communication

Fig. 3 shows that all network communications in Peacock happen in the form of RPCs. The largest fraction of *communication cost* lies in WorkWithSampler, where data servers send data blocks to sampling servers, and wait for responses containing updated topic assignment $\boldsymbol{z}^m_{ivd}$.

We reduce the communication cost using *pipeline* techniques. To avoid the overflow of network communication buffer, the data server sends just a few document fragments known as a *package* rather than sending a block in an RPC to the sampling server. Instead of waiting the response from the sampling server before sending the next package, the data server sends T packages concurrently. On the sampling server, there are multiple *goroutines*, a kind of light-weighted

```
1: procedure SampleSegment(segment)
2:   par-for m ← 1..M
3:     data-server[m].LoadShard(segment, m)
4:   par-for m ← 1..M
5:     for dig ← 1..M do                ▷ dig indices diagonals
6:       data-server[(m+dig)%M].WorkWithSampler(m)
7:   par-for m ← 1..M
8:     Ψ_k ← Ψ_k+ sampling-server[m].GetDiffNt
9:   par-for m ← 1..M
10:    sampling-server[m].SetNt(Ψ_k)
11:  par-for m ← 1..M
12:    data-server[m].SaveShard(segment, m)
13:  par-for m ← 1..M
14:    Ω_kn ← Ω_kn+ data-server[m].CountNtn
```

Figure 4: Faster sampling of corpus segments.

thread scheduled by the Go runtime system, to process these packages and respond to the data server. The data server maintains a data structure with T slots, and each keeps track of an out-going package. The data server clears a slot after receiving a response of the corresponding package, or getting a timeout. Once there are empty slots and packages to be processed, the data server would continue sending packages. In general, this pipeline optimizes the throughput by overlaps the sending, processing and responding of packages.

Maximizing the throughput depends on finding the optimal configuration of two parameters: package size L and pipeline capacity T. The product, $L \times T$, is proportional to the size of memory used as communication buffer. In practice, there would be an upper limit of buffer size, c, and we would make full use of it to get the maximum throughput. This can be written as the constraint function, $L \times T = c$, which is a curve on the two dimensional space L and T. The best configuration would be a point on this curve. In our computing environment, a practical c value is 200MB. The measure of time consumption with respect to the curve $L \times T = c$ is shown in Table 1, where the time consumption is larger at both ends of this curve, and the optimal configuration lies in the middle of the curve.

3.1.3 Lock-free Synchronization

After issuing parallel executions of the loop body in Fig. 3, the **par-for** does a synchronization operation that waits for the completions of all executions of sampling servers called *synchronization lock problem*. We address this by three lock-free strategies. First, to avoid data skewness in each data block, we randomly shuffle data by rows and columns so that each block contains almost equal number of word tokens in practice [32].

Second, we can further reduce the synchronization cost of the **par-for** on line 4 of Fig. 4 by balancing workload of sampling servers. Because each sampling server processes a column of corpus blocks, it is desirable that block columns contain similar word frequencies. This can be achieved using a pre-training scheduler which assigns vocabulary words to Φ^m_{vk} shards. As with PLDA+ [17], we use the weighted round-robin method for this word assignment. We first sort vocabulary words in descending order by their frequency, and pick the word with the largest frequency and assign it to the Φ^m_{vk} shard with the accumulative word frequency. Then, we update the accumulated word frequency of Φ^m_{vk}. This placement process is repeated until all words have been assigned. Weighted round-robin has been empirically shown to achieve a balanced load with a high probability [5].

Finally, in Fig. 3, the nested loop starting from line 10 invokes the **par-for** many times and introduces many waits. The problem can be relieved by swapping the inner and outer loop as shown in Fig. 4. In this change, two **par-for** structures at line 13 and line 15 are moved one upper level. This relaxes the aggregation and redistribution of vector Ψ_k from a per-diagonal granularity to a per-segment granularity. This relaxed aggregation does not affect the correctness of the distributed GS algorithm within the stochastic framework [30]. After swapping the **par-for** at line 11 with its outer loop at line 10, a data server might work with more than one sampling server simultaneously. However, this would introduce conflicts in accessing the $\boldsymbol{z}^m_{ivd}$ shard maintained by the data server. We address this conflict problem by two methods. First, we use $M+1$ data servers as shown in Fig. 2 layer 1 denoted by "Free". These "Free" data servers provide additional conflict-free data blocks for computing without waiting for the completion of other sampling servers. The coordinator will schedule the finished sampling server to those conflict-free data blocks having the minimum number of visits. In this way, we assure that all data blocks will have almost equal number of visits. Second, each data server maintains two replica of the $\boldsymbol{z}_{ivd}$ shard: $\boldsymbol{z}^{old}_{ivd}$ and $\boldsymbol{z}^{new}_{ivd}$ without conflicts. After receiving the response of updated package from the sampling server, the data server applies the difference between the response and $\boldsymbol{z}^{old}_{ivd}$ to $\boldsymbol{z}^{new}_{ivd}$.

3.1.4 Fault Recovery

Data parallelism also helps fault recovery, which is critical in large-scale machine learning. Consider that a parallel learning job may take days or even weeks, it is very probable that some workers fail or be preempted during the period. If the system cannot recover when it fails, we would have to restart the job from the beginning. Since the restart might fail again, the learning job would never finish. As the layer 1 configurations work independently within every few iterations, it is straightforward to restart any failed configuration based on the independent checkpoint on hard disks. The restart can be implemented simply using the SSH command, or sophisticated cluster management systems like Apache YARN. This architecture is similar to Google's architecture for parallel deep learning [9], which refers to configurations as models. More than achieving fine-grained fault recovery, this design also improves the parallelism and makes Peacock highly scalable.

3.2 Real-time Prediction

It is critical in online applications like search engines and online advertising system to predict latent semantics of new user queries in real-time based on the large number of topics. In typical Internet services, the response time of back-end servers is measured in milliseconds. Given LDA models with at least 10^5 topics, few inference algorithms are efficient enough to do real-time prediction. Hence, we propose a real-time inference algorithm, RT-LDA, especially for prediction of new queries.

The basic idea of RT-LDA is to replace the sampling operation in SparseLDA [26] by the max operation. This makes RT-LDA a hill climbing algorithm whose search path consists of line segments aligned with axes of the topic space,

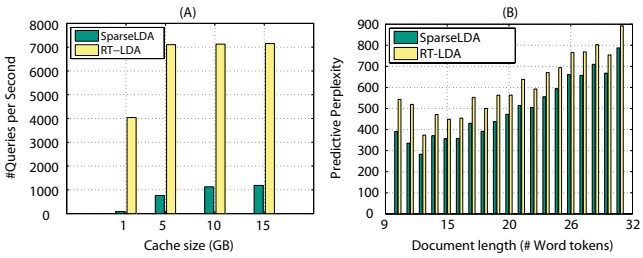

Figure 5: Comparisons between RT-LDA and SparseLDA [26] in (A) speed and (B) accuracy.

like the CDN (coordinate descending using one-dimensional Newton step) algorithm [27] widely used in learning regression models. The max operation in RT-LDA can be optimized using a cache-based technique. According to Eq. (1), the max operator in RT-LDA is

$$\begin{aligned} &\max_{k\in[1,K]} P(z_{ivd}=k \mid \boldsymbol{z}_{\neg ivd}, \boldsymbol{x}_{\neg ivd}, x_{ivd}=1, \alpha, \beta) \\ =&\max_{k\in[1,K]} \hat{P}(v|k)(\Theta_{kd}+\alpha_k) \\ =&\max_{k\in[1,K]} \hat{P}(v|k)\Theta_{kd} + \hat{P}(v|k)\alpha_k, \end{aligned} \tag{2}$$

where $\hat{P}(v|k)$ is the empirical probability matrix computed by normalizing columns of $\Phi_{V\times K}$. In prediction, $\hat{P}(v|k)$ and α_k are constants, whereas Θ_{kd} changes with the updating of topic assignments. This makes it viable to precompute $\max_k \hat{P}(v|k)\alpha_k$, whose result is a sparse matrix R,

$$R_{vk} = \begin{cases} \hat{P}(v|k)\alpha_k & \text{if } k = \max_{k'} \hat{P}(v|k')\alpha_{k'}, \\ 0 & \text{otherwise.} \end{cases} \tag{3}$$

We save R in a compact data structure which contains only V non-zero elements. The compact R is an approximation to Eq. (2), where the error of the approximation is caused by non-zero elements in Θ_{kd}. We rewrite Eq. (2) as

$$\begin{aligned} &\max_{k\in[1,K]} P(z_{ivd}=k \mid \boldsymbol{z}_{\neg ivd}, \boldsymbol{x}_{\neg ivd}, x_{ivd}=1, \alpha, \beta) \\ =&\max_k \left[R^*_{vk}, \max_{\substack{k\in[1,K] \\ s.t. \Theta_{kd}>0}} \hat{P}(v|k)(\Theta_{kd}+\alpha_k) \right], \end{aligned} \tag{4}$$

where R^*_{vk} denotes the non-zero element in the column of R corresponding to vocabulary word v. Different from the max operation in Eq. (2), which iterates over $k \in [1, K]$, the first max operation in Eq. (4) compares two values, and the second max operation visits only non-zero elements in Θ_{kd}. Suppose that the maximum number of non-zero elements in a query d is the length of the query, Eq. (4) makes RT-LDA significantly faster than SparseLDA when the number of topics $K \geq 10^5$.

Fig. 5A compares the prediction speed between RT-LDA and SparseLDA [26]. We use the the number of query-per-second (QPS) of RT-LDA and SparseLDA on a real backend inference server with 100% CPU load. The x-axis is the cache size. Generally, larger cache leads to faster prediction, but the performance reaches the upper bound with the increase of cache size in Gigabytes (GB). This setting implies that the response time of prediction is the reciprocal of the QPS. We see that RT-LDA is about an order of magnitude faster than SparseLDA. Fig. 5B compares RT-LDA and SparseLDA for their topic modeling accuracy measured in predictive perplexity [29], which is a standard performance measure for topic modeling accuracy. The lower perplexity means a higher topic modeling accuracy on new test data set. This experiment uses 1,200 Wikipedia titles by clustering them into 20 groups with various length. For all these groups, we randomly select 10% as test data set and retain the remaining 90% as training set. We see that the accuracy of the two algorithms, measured in perplexity, are very close. RT-LDA loses some tolerable topic modeling accuracy to get a faster speed than SparseLDA. We can further improve the effectiveness of RT-LDA by running multiple line searches in parallel and then averaging results of all these parallel trails. Because RT-LDA is much more efficient than SparseLDA, the Peacock system can afford many parallel trails to extract topic features from new queries.

3.3 Topic De-Duplication

Although topic duplication has been rarely discussed in previous literature, it becomes a main challenge in topic feature engineering. As noted in [23], when learning LDA, frequent words often dominate more than one topics, and the learned topics are similar to each other. We refer to these similar topics as *duplicates*. Generally, when learning $\geq 10^5$ topics, around 20% $\sim$ 40% topics have duplicates in practice. If a query d is 60% about the topic A and 40% about the topic B, the query d is mainly about the topic A. However, if a topic A has three duplicates, $A1$, $A2$ and $A3$, the GS algorithm would follow Eq. (1) to scatter the 60% weight of A in d to $A1$, $A2$ and $A3$. If each duplicate gets 1/3 of the original 60% topic A, the interpretation of the query would become mainly about topic B, though the truth is that the query is mainly about topic A.

We remove topic duplicates by two methods. First, we learn asymmetric Dirichlet priors over the document-topic distributions [23], which substantially increases the robustness of LDA to variations in the number of topics and to the highly skewed word frequency distributions common in natural language. Asymmetric priors over document-topic distributions automatically combine similar topics into one large topic, rather than splitting topics more uniformly by symmetric priors. In practice, we can set a very large $K = 10^6$ value at initial learning time, and prune duplicates by asymmetric Dirichlet priors. Those topics with very small Dirichlet priors would be automatically weighted trivial by RT-LDA (Section 3.2) at serving time. We find this approach prevents common words from dominating many topics, thus leaves the room for long-tail topics. Second, we cluster topic duplicates if their L_1-distance is below a threshold. The lower L_1-distance threshold means that we would remove more duplicates from large number of topics.

4. EMPIRICAL STUDIES IN BIG DATA

We evaluate Peacock's topic modeling performance for big data by three performance measures: 1) Speedup: how much faster than a sequential GS algorithm; 2) Scalability: the ability to handle a growing number of topics; 3) Accuracy: the log-likelihood of LDA achieved by increasing number of iterations [22, 2]. The baseline is the state-of-the-art industrial solution Yahoo!LDA [22, 2] with open source codes[1]. Likewise, Yahoo!LDA also distributes SparseLDA [26] over

[1] https://github.com/sudar/Yahoo_LDA

multiple machines but using a shared memory environment based on the *memcached* technique.

4.1 Data Sets

For a fair comparison, we use the same publicly available data set PUBMED[2], which contains 8.2 million documents with an average length of around 90 word tokens each. The vocabulary size of PUBMED is 1.4×10^5. We also compose the training corpus of search queries received in recent months called SOSO. The pre-processing of the corpus contains five steps: 1) Transform each query into word tokens, and count word frequencies. 2) Remove those words with low frequency, which are likely typos. 3) Remove those words with very high frequency, because common words tend to dominate all topics [23]. 4) De-duplicate queries: If a query appears multiple times, we keep only one appearance in corpus. This allows us to include a large variety of user intentions within a certain amount of training corpus. This also lowers the weight of frequent queries in the corpus. 5) Remove those queries containing only one word, because single-word queries do not provide word co-occurrence counts, which is a clue used by LDA to infer topics. The processed corpus contains one billion search queries with 4.5 word tokens per query in average and takes 17.2GB storage space. The vocabulary size of SOSO is around 2.1×10^5. Obviously, SOSO is around 6 times larger than PUBMED.

4.2 Results

Fig. 6 shows the speedup performance (fixing $K = 1000$), where the x-axis is the number of cores and y-axis is the training time ratio of 100 cores over the time of other number of cores in x-axis. We see that Peacock on average achieves around 4.2 speedup when the number of cores is 1000, which implies that the communication and synchronization in Peacock take about half of the training time. Yahoo!LDA has a much better speedup than Peacock when the number of cores is small (≤ 1000). However, its speedup performance drops significantly when the number of cores increases from 1000 to 3000. The possible reason is that Yahoo!LDA does not consider the lock-free synchronization problem. In practice, the very large number of cores will cause longer waiting time when accessing the shared memory based on *memcached* technique. Peacock scales much better to the large number of cores by pipeline communication and lock-free synchronization (Sections 3.1.2 and 3.1.3). We see that Peacock is slower than Yahoo!LDA when the number of cores is small. The reason is that we use more separate sampling servers for model parallelism leading to the additional communication and synchronization costs, which remains almost a constant ratio in training time by pipeline techniques.

Fig. 6 also shows the training time per iteration with the growth of number of topics (fixing the number of cores 500), $K = \{10^2, 10^3, 10^4, 10^5\}$. With K ranging from 10^2 to 10^4, the corresponding increase of training time of Peacock is small. When K increases by 10 times from 10^4 to 10^5, the increase of training time is close to linear. The scalability of Peacock with respect to K comes mainly from the model and data parallelism (Section 4) resulting in very fast sampling performance with a small memory footprint. From $K = 10^4$ to $K = 10^5$, Peacock uses significantly more training time because the topic sparseness of each word token becomes

[2] http://archive.ics.uci.edu/ml/datasets/Bag+of+Words

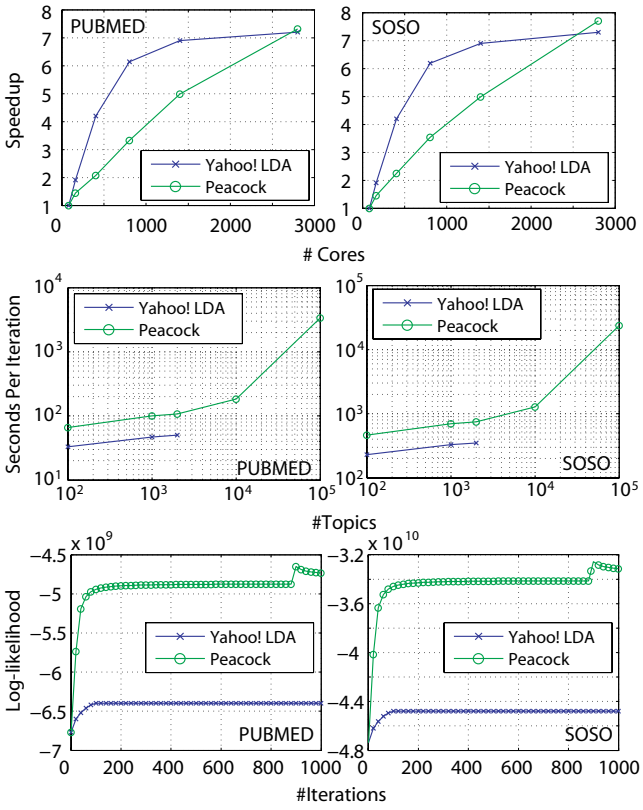

Figure 6: Comparisons between Peacock and Yahoo!LDA [22, 2] in big data and big model.

lower in SparseLDA [26]. As a comparison, Yahoo!LDA occurs out of memory problem when the the number of topics $K \geq 10^4$ because it does not consider storing a big topic-word matrix Φ_{vk}. To solve this problem, Peacock divides this matrix into small model shards for large-scale number of topics. Since Peacock uses additional communication to synchronize more sampling servers, its per iteration training time is higher than that of Yahoo!LDA.

Finally, Fig. 6 shows the log-likelihood (the higher the better model quality) curves by iterations (fixing $K = 1000$ and number of cores 500). We observe that Peacock has a rise after 900 iterations because we start asymmetric prior optimization and topic de-duplication (Section 3.3), which can improve the topic model quality. Although both Peacock and Yahoo!LDA use SparseLDA [26], Peacock converges to a higher log-likelihood level. The reason is partly because Yahoo!LDA uses the approximate synchronization to speedup its performance leading to a slightly worse model quality, which has been also observed in their own work [22, 2].

To summarize, we see that Yahoo!LDA is more efficient for small-scale ($K \approx 10^3$) topic modeling tasks, while Peacock is more suitable for solving large-scale ($K \geq 10^5$) topic modeling problems in industrial applications.

5. ONLINE APPLICATIONS

After Peacock learns $K \geq 10^5$ topics from $D \geq 10^9$ queries, we need to integrate the topic features into existing search engine and online advertising system. We extract topic

features from new queries based on the topic distribution $P(v|k) = \Phi_{vk}$ learned by Peacock. Give the word tokens of a new query d, we use RT-LDA to predict its topic distribution $P(k|d)$ by fixing $P(v|k)$. Employing the Bayes' rule, we calculate the likelihood of a vocabulary word v given a query d,

$$P(v|d) = \sum_{k=1}^{K} P(v|k)P(k|d). \quad (5)$$

The V-length vector $P(v|d)$ is compatible with the standard word vector space model. If we rank $P(v|d)$ in descending order, we obtain top likely topic features in the input query.

5.1 Experimental Settings

Search engines use the well-known vector space model in information retrieval and computes cosine similarity between queries and documents in their vectors representations. We accelerate this process by using the Weak-AND algorithm [8]. Peacock replaces the word vector features of each query by top 30 likely topic features (5) (Top 30 largest values from V-length vector $P(v|d)$) inferred by RT-LDA in the head of each posting list used by the Weak-AND algorithm, which makes the query-document similarity computing efficient enough to be deployed in a real search engine.

Online advertising has been an fundamental financial support of the many free Internet services [7]. Most contemporary online advertising systems follow the Generalized Second Price (GSP) auction model [11], which requires that the system is able to predict the click-through rate (pCTR) of an ad, where pCTR is an important clue in GSP to ranking ads and pricing clicks. One of the key questions with the pCTR is the availability of suitable input features or predictor variables that allow accurate CTR prediction for a given ad impression [21]. These features can be grouped into three categories: Ad features including bid phrases, ad title, landing page, and a hierarchy of advertiser account, campaign, ad group and ad creative. User features include recent search queries, and user behavior data. Context features include display location, geographic location, content of page under browsing, and time. Most of these features are text data in word vector space [13]. We learn an L_1-regularized log-linear model [3] as the baseline for pCTR, which uses a set of text and other features such as ad title, content of page under browsing, content of landing page, ad group id, demographic information of users, categories of ad group and categories of the page under browsing. As a comparison, Peacock adds all topic features (5), i.e., the V-length topic feature vector $P(v|d)$, in baseline text and other features as input to L_1-regularized log-linear model [3].

In both applications, the training data set of Peacock is SOSO described in subsection 4.1. For Peacock, we set two layer 1 configurations with 1 coordinator server and $M = 125$ aggregation servers, where each configuration contains $M = 125$ sampling servers and $M + 1 = 126$ data servers.

5.2 Results

For information retrieval, our test-bed is a real search engine, `www.soso.com`, which ranks the fourth largest in China market. The test data is used for routinely relevance evaluation, containing $4,818$ randomly selected queries and $121,588$ query-URL pairs with human labeled relevance rate. Every query-URL pair was rated by three human editors and the average rate was taken. We compute mean average precision (MAP) using TREC evaluation tool [1]. The higher MAP means the better retrieval performance. Fig. 7A shows that the topic features improve the relevance measure by MAP. The relevance improvement grows steadily with the increasing number of topics from 10^2 to 10^5. However, the growth of MAP becomes less salient when the number of topics changes from 10^4 to 10^5. This is mainly attributed to the problem of *topic duplication*. Fig. 7B shows that topic de-duplication method can further improve the relevance of information retrieval. The MAP value of retrieval grows when we prune more duplicated topics (lower L_1 distance can prune more similar topics in Section 3.3). Usually, the MAP stops increasing when we prune duplicates from the initial 10^6 to around 10^5 topics. This result implies that 10^5 is a critical number of topics to describe subtle word senses in big query data with 2.1×10^5 vocabulary words.

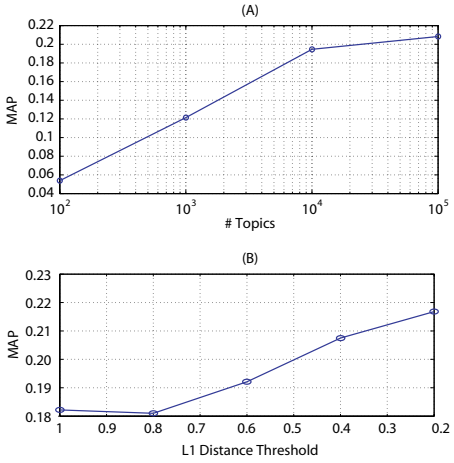

Figure 7: (A) Topic features improve retrieval in search engine. (B) Performance improvement in retrieval after topic de-duplication by topic clustering based on L_1 distance.

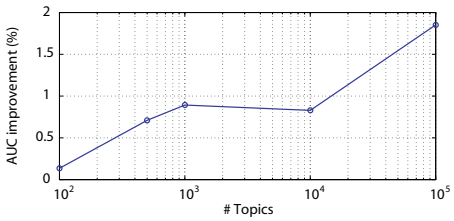

Figure 8: Topic features improve the pCTR performance in online advertising system.

The online advertising experiment is conducted on a real contextual advertising system, `https://tg.qq.com/`. This system logs every ad shown to a particular user in a particular page view as an *ad impression*. It also logs every click of an ad. By taking each impression as a training instance, and labeling it by whether it was clicked, we obtain 9.9 billion training instances and 1.1 billion test instances. We train 5 *hypothetical models*, whose topic features are extracted using 5 different LDA models with 10^2, 5×10^2, 10^3, 10^4 and 10^5 topics, respectively. Following the judgment rule of Task 2

in KDD Cup 2012, a competition of ad pCTR, we compare our hypothetical models with the baseline by their prediction performance measured in area under the curve (AUC). Fig. 8 shows that all hypothetical models gain relative AUC improvement (%) than the baseline (AUC = 0.7439). This verifies the value of big LDA models. Also, the AUC improvement grows with the increase of the number of topics learned by Peacock. The reason that the performance of 10^4 is lower than that of 10^3 is because of many topic duplicates in 10^4 topics. After using automatic topic de-duplication by asymmetric Dirchlet prior learning (Section 3.3), the performance of 10^5 becomes better than that of 10^4. This result is consistent with those in Fig. 7.

6. CONCLUSIONS

Topic modeling techniques for big data are needed in many real-world applications. In this paper, we confirm that a big LDA model with at least 10^5 topics inferred from 10^9 search queries can achieve a significant improvement in industrial applications like search engines and online advertising system. We propose a unified solution Peacock to do topic modeling for big data. Peacock uses a hierarchical parallel architecture to handle large-scale data as well as LDA parameters. In addition, Peacock addresses some novel problems in big topic modeling, including real-time prediction and topic de-duplication. We show that Peacock is scalable to more topics than the current state-of-the-art industrial solution Yahoo!LDA. Through two online applications, we also obtain the following conclusions:

- The good performance is often achieved when the number of topics is approximately equal to or more than the number of vocabulary words. In our experiments, the vocabulary size is 2.1×10^5 so that the number of topics $K \geq 10^5$. In other industrial applications, the vocabulary size may reach a few millions or even a billion. The Peacock system can do topic feature learning when $K \geq 10^7$ is needed.

- Topic de-duplication is a key technical component to ensure that $K \geq 10^5$ topics can provide high-quality topic features. Better topic de-duplication techniques remain to be an open research issue.

- Real-time topic prediction method for a large number of topics is also important in industrial applications. If $K \geq 10^7$, faster prediction methods are needed and remain to be a future research issue.

7. ACKNOWLEDGEMENTS

This work is supported by National Grant Fundamental Research (973 Program) of China under Grant 2014CB340304, NSFC (Grant No. 61373092 and 61033013), Natural Science Foundation of the Jiangsu Higher Education Institutions of China (Grant No. 12KJA520004), and Innovative Research Team in Soochow University (Grant No. SDT2012B02).

8. REFERENCES

[1] Trec evaluation tool, `http://trec.nist.gov/trec_eval/`.

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

[3] G. Andrew and J. Gao. Scalable training of L1-regularized log-linear models. In *ICML*, 2007.

[4] A. Asuncion, P. Smyth, and M. Welling. Asynchronous distributed learning of topic models. In *NIPS*, 2008.

[5] P. Berenbrink, T. Friedetzky, Z. Hu, and R. Martin. On weighted balls-into-bins games. *Theor. Comput. Sci.*, 409(3):511–520, 2008.

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

[7] A. Broder and V. Josifovski. Lecture introduction to computational advertising. Stanford University, Computer Science, Online Lecture Notes.

[8] A. Z. Broder, D. Carmel, M. Herscovici, A. Soffer, and J. Zien. Efficient query evaluation using a two-level retrieval process. In *CIKM*, 2003.

[9] J. Dean, G. S. Corrado, R. Monga, K. Chen, M. Devin, Q. V. Le, M. Z. Mao, M. Ranzato, A. Senior, P. Tucker, K. Yang, and A. Y. Ng. Large scale distributed deep networks. In *NIPS*, 2012.

[10] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[11] B. Edelman, M. Ostrovsky, and M. Schwarz. Internet advertising and the generalized second-price auction: Selling billions of dollars worth of keywords. *American Economic Review*, 2007.

[12] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. In *KDD*, pages 69–77, 2011.

[13] T. Graepel, J. Q. Candela, T. Borchert, and R. Herbrich. Web-scale bayesian click-through rate prediction for sponsored search advertising in microsoftâĂŹs bing search engine. In *ICML*, 2010.

[14] D. Graff and C. Cieri. English Gigaword, 2003.

[15] T. Griffiths and M. Steyvers. Finding scientific topics. *PNAS*, 101:5228–5235, 2004.

[16] H. Li, Y. Wang, D. Zhang, M. Zhang, and E. Chang. PFP: Parallel FP-growth for query recommendation. In *ACM RecSys*, 2008.

[17] Z. Liu, Y. Zhang, E. Chang, and M. Sun. PLDA+: Parallel latent Dirichlet allocation with data placement and pipeline processing. *ACM Trans. Intelligent Systems and Technology*, 2011.

[18] T. Minka and J. Lafferty. Expectation-propagation for the generative aspect model. In *UAI*, 2002.

[19] D. Newman, A. Asuncion, P. Smyth, and M. Welling. Distributed inference for latent Dirichlet allocation. In *NIPS*, 2007.

[20] D. Newman, J. H. Lau, K. Grieser, and T. Baldwin. Automatic evaluation of topic coherence. In *ACL*, 2010.

[21] M. Richardson, E. Dominowska, and R. Ragno. Predicting clicks: Estimating the click-through rate for new ads. In *WWW*, 2007.

[22] A. Smola and S. Narayanamurthy. An architecture for parallel topic models. In *VLDB*, 2010.

[23] H. Wallach, D. Mimno, and A. McCallum. Rethinking LDA: Why priors matter. In *NIPS*, 2009.

[24] Y. Wang, H. Bai, M. Stanton, W. Chen, and E. Chang. PLDA: Parallel latent Dirichlet allocation for large-scale applications. In *AAIM*, 2009.

[25] F. Yan and N. Xu. Parallel inference for latent Dirichlet allocation on graphics processing units. In *NIPS*, 2009.

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

[27] G.-X. Yuan, K.-W. Chang, C.-J. Hsieh, and C.-J. Lin. A comparison of optimization methods and software for large-scale L1-regularized linear classification. *Journal of Machine Learning Research*, 11:3183–3234, 2010.

[28] J. Zeng. A topic modeling toolbox using belief propagation. *J. Mach. Learn. Res.*, 13:2233–2236, 2012.

[29] J. Zeng, W. K. Cheung, and J. Liu. Learning topic models by belief propagation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 35(5):1121–1134, 2013.

[30] J. Zeng, Z.-Q. Liu, and X.-Q. Cao. Online belief propagation for topic modeling. *arXiv:1210.2179 [cs.LG]*, 2012.

[31] K. Zhai, J. Boyd-Graber, N. Asadi, and M. Alkhouja. Mr. LDA: A flexible large scale topic modeling package using variational inference in mapreduce. In *WWW*, 2012.

[32] Y. Zhuang, W.-S. Chin, Y.-C. Juan, and C.-J. Lin. A fast parallel sgd for matrix factorization in shared memory systems. In *Recsys*, 2013.