Computing Web-scale Topic Models using an Asynchronous Parameter Server

Rolf Jagerman
University of Amsterdam
Amsterdam, The Netherlands
rolf.jagerman@uva.nl

Carsten Eickhoff
ETH Zürich
Zürich, Switzerland
carsten.eickhoff@inf.ethz.ch

Maarten de Rijke
University of Amsterdam
Amsterdam, The Netherlands
derijke@uva.nl

ABSTRACT

Topic models such as Latent Dirichlet Allocation (LDA) have been widely used in information retrieval for tasks ranging from smoothing and feedback methods to tools for exploratory search and discovery. However, classical methods for inferring topic models do not scale up to the massive size of today's publicly available Web-scale data sets. The state-of-the-art approaches rely on custom strategies, implementations and hardware to facilitate their asynchronous, communication-intensive workloads.

We present APS-LDA, which integrates state-of-the-art topic modeling with cluster computing frameworks such as Spark using a novel *asynchronous* parameter server. Advantages of this integration include convenient usage of existing data processing pipelines and eliminating the need for disk writes as data can be kept in memory from start to finish. Our goal is not to outperform highly customized implementations, but to propose a general high-performance topic modeling framework that can easily be used in today's data processing pipelines. We compare APS-LDA to the existing Spark LDA implementations and show that our system can, on a 480-core cluster, process up to 135× more data and 10× more topics without sacrificing model quality.

CCS CONCEPTS

•Information systems →Document representation; Document topic models;

ACM Reference format:
Rolf Jagerman, Carsten Eickhoff, and Maarten de Rijke. 2017. Computing Web-scale Topic Models using an Asynchronous Parameter Server. In *Proceedings of SIGIR '17, August 07-11, 2017, Shinjuku, Tokyo, Japan,* , 4 pages.
DOI: http://dx.doi.org/10.1145/3077136.3084135

1 INTRODUCTION

Probabilistic topic models are a useful tool for discovering a set of latent themes that underlie a text corpus [2, 6]. Each topic is represented as a multinomial probability distribution over a set of words, giving high probability to words that co-occur frequently and small probability to those that do not.

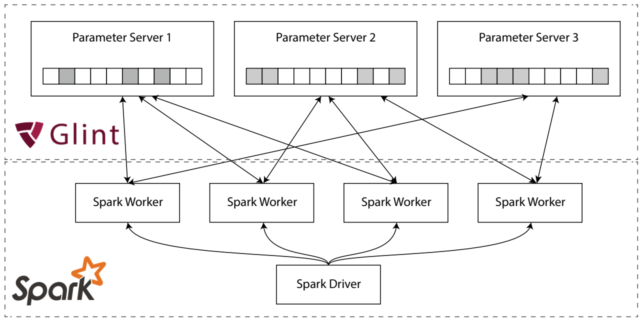

Figure 1: High-level overview of the Glint parameter server architecture and its interaction with Spark. The parameter servers provide a distributed and concurrently accessed parameter space for the model being learned.

Recent information retrieval applications often require very large-scale topic modeling to boost their performance [13], where many thousands of topics are learned from terabyte-sized corpora. Classical inference algorithms for topic models do not scale well to very large data sets. This is unfortunate because, like many other machine learning methods, topic models would benefit from a large amount of training data.

When trying to compute a topic model on a Web-scale data set in a distributed setting, we are confronted with a major challenge:

How do individual machines keep their model synchronized?

To address this issue, various distributed approaches to LDA have been proposed. The state-of-the-art approaches rely on custom strategies, implementations and hardware to facilitate their asynchronous, communication-intensive workloads [3, 12, 13]. These highly customized implementations are difficult to use in practice because they are not easily integrated in today's data processing pipelines.

We propose APS-LDA, a distributed version of LDA that builds on a widely used cluster computing framework, Spark [14]. The advantages of integrating model training with existing cluster computing frameworks include convenient usage of existing data-processing pipelines and eliminating the need for intermediate disk writes since data can be kept in memory from start to finish [10]. However, Spark is bound to the typical map-reduce programming paradigm. Common inference algorithms for LDA, such as collapsed Gibbs sampling, are not easily implemented in such a paradigm because they rely on a large mutable parameter space that is updated concurrently. We address this by adopting the parameter server model [9], which provides a distributed and concurrently accessed parameter space for the model being learned (see Fig. 1).

2 DISTRIBUTED LDA

We present APS-LDA, our distributed version of LDA, which builds on the LightLDA algorithm [13]; it uses an asynchronous version of the parameter server, as we will detail in Section 3.

2.1 LightLDA

LightLDA performs a procedure known as collapsed Gibbs sampling, which is a Markov Chain Monte-Carlo type algorithm that assigns a topic $z \in \{1, \ldots, K\}$ to every token in the corpus. It then repeatedly re-samples the topic assignments z. The LightLDA algorithm provides an elegant method for re-sampling the topic assignments in $O(1)$ time by using a Metropolis-Hastings sampler. This is important because sampling billions of tokens is computationally infeasible if every sampling step would use $O(K)$ operations, where K is a potentially large number of topics.

To re-sample the topic assignments z, the algorithm needs to keep track of the statistics n_k, n_{wk} and n_{dk}:

- n_k: Number of times any word was assigned topic k
- n_{wk}: Number of times word w was assigned topic k
- n_{dk}: Number of times a token in document d was assigned topic k

It is clear that the document-topic counts n_{dk} are document-specific and thus local to the data and need not be shared across machines. However, the word-topic counts n_{wk} and topic counts n_k are global and require sharing. The parameter server provides a shared interface to these values in the form of a distributed matrix storing n_{wk}, and a distributed vector storing n_k.

2.2 APS-LDA: A Re-design of LightLDA

Despite its attractive properties, LightLDA has an important shortcoming. It uses a stale-synchronous parameter server in which push requests are batched together and sent once when the algorithm finishes processing its current partition of the data. This architecture uses a fixed network thread and may cause a stale model, where individual machines are unable to see updates from other machines for several iterations.

In contrast, our approach sends push requests *asynchronously* during the compute stage. These more frequent but smaller updates have a number of essential advantages:

(1) It decreases the staleness of the model while it is computing. With our approach it is possible to see updates from other machines within the same iteration over the data, something that is not possible with the standard parameter server.
(2) It makes mitigating network failure easier as small messages can be resent more efficiently.
(3) It enables the algorithm to take advantage of more dynamic threading mechanisms such as fork-join pools and cached thread pools [11].

The move from such a fixed threaded design to a fully asynchronous one requires a re-design of LightLDA. Algorithm 1 describes the APS-LDA method. At the start of each iteration, the algorithm performs a synchronous pull on each processor p to get access to the global topic counts n_k. It then iterates over the vocabulary terms, and asynchronously pulls the word-topic counts n_{wk} (line 6). These asynchronous requests call back the RESAMPLE procedure when they complete. The RESAMPLE procedure (line 12) starts by

Algorithm 1 APS-LDA: Asynchronous Parameter Server LDA.

```
1:  $\mathcal{P} \leftarrow$ Set of processors,
2:  $\mathcal{D} \leftarrow$ Collection of documents,
3:  $\mathcal{V} \leftarrow$ Set of vocabulary terms

4:  for $p \in \mathcal{P}$ in parallel do
5:      $\mathcal{D}_p \subseteq \mathcal{D}$
6:      $n_k \leftarrow$ SyncPull($\{n_k \mid k = 1 \ldots K\}$)
7:      for $w \in \mathcal{V}$ do
8:          on AsyncPull($\{n_{wk} \mid k = 1 \ldots K\}$)
9:              call Resample($\mathcal{D}_p, n_{wk}, n_k$)
10:     end for
11: end for

12: procedure Resample($\mathcal{D}_p, n_{wk}, n_k$)
13:     $a \leftarrow$ AliasTable($n_{wk}$)
14:     for $(w, z_{\text{old}}) \in d \in \mathcal{D}_p$ do
15:         $z_{\text{new}} \leftarrow$ MetropolisHastingsSampler($a, d, w, z_{\text{old}}, n_k, n_{wk}$)
16:         AsyncPush($\{n_{wk} \leftarrow n_{wk} + 1\}$) for $k = z_{\text{new}}$
17:         AsyncPush($\{n_k \leftarrow n_k + 1\}$) for $k = z_{\text{new}}$
18:         AsyncPush($\{n_{wk} \leftarrow n_{wk} - 1\}$) for $k = z_{\text{old}}$
19:         AsyncPush($\{n_k \leftarrow n_k - 1\}$) for $k = z_{\text{old}}$
20:     end for
21: end procedure
```

computing an alias table on the available word-topic counts n_{wk}. This alias table is a datastructure that can sample from a categorical probability distribution in amortized $O(1)$ time. The algorithm then iterates over the local partition of the data $\mathcal{D}_p$ where it resamples every (token, topic) pair using LightLDA's $O(1)$ Metropolis-Hastings sampler, which requires the earlier mentioned alias table. Changes to the topic counts are pushed asynchronously to the parameter server while it is computing (Lines 16 to 19)

Note that all of our push requests either increment or decrement the counters n_{wk} and n_k. The parameter server exploits this fact by aggregating these updates via addition, which is both commutative and associative. This eliminates the need for complex locking schemes that are typical in key-value storage systems. Instead, the updates can be safely aggregated through an atomic integer structure that is easy to implement.

In the next section, we will discuss the asynchronous parameter server that makes the implementation of this algorithm possible.

3 PARAMETER SERVER ARCHITECTURE

The traditional parameter server architecture [8] is a complete machine learning framework that couples task scheduling, a distributed (key, value) store for the parameters and user-defined functions that can be executed on workers and servers. As a result, there is considerable complexity in the design, setup and implementation of a working parameter server, making it difficult to use in practice.

We present Glint,[1] an open-source asynchronous parameter server implementation. Our implementation is easily integrated with the cluster computing framework Spark, which allows us to leverage Spark features such as DAG-based task scheduling, straggler mitigation and fault tolerance. This integration is realized

[1]https://github.com/rjagerman/glint/

by decoupling the components of the traditional parameter server architecture and removing the dependency on task scheduling. This is accomplished by simplifying the parameter server interface to a set of two operations:

(1) **Asynchronously 'Pull' data from the servers.**
This will query parts of the matrix or vector.
(2) **Asynchronously 'Push' data to the servers.**
This will update parts of the matrix or vector.

The goal of our parameter server implementation is to store a large distributed matrix and provide a user with fast queries and updates to this matrix. In order to achieve this, it will partition and distribute the matrix to multiple machines. Each machine only stores a subset of rows. Algorithms interact with the matrix through the *pull* and *push* operations, unaware of the physical location of the data.

3.1 Pull action

Whenever an algorithm wants to retrieve entries from the matrix it will call the *pull* method. This method triggers an asynchronous pull request with a specific set of row and column indices that should be retrieved. The request is split up into smaller requests based on the partitioning of the matrix such that there will be at most one request per parameter server.

Low-level network communication provides an 'at-most-once' guarantee on message delivery. This is problematic because it is impossible to know whether a message sent to a parameter server is lost or just takes a long time to compute. However, since pull requests do not modify the state of the parameter server, we can safely retry the request multiple times until a successful response is received. To prevent flooding the parameter server with too many requests, we use an exponential back-off timeout mechanism. Whenever a request times out, the timeout for the next request is increased exponentially. If after a specified number of retries there is still no response, we consider the pull operation failed.

3.2 Push action

In contrast to pull requests, a *push* request will modify the state on the parameter servers. This means we cannot naïvely resend requests on timeout because if we were to accidentally process a push request twice it would result in a wrong state on the parameter server. We created a hand-shaking protocol to guarantee 'exactly-once' delivery on push requests.[2] The protocol first attempts to obtain a unique transaction *id* for the push request. Data is transmitted together with the transaction *id*, allowing the protocol to later acknowledge receipt of the data. A timeout and retry mechanism is only used for messages that are guaranteed not to affect the state of the parameter server. The result is that pushing data to the parameter servers happens exactly once.

3.3 LDA implementation

We have implemented the APS-LDA algorithm using Spark and the asynchronous parameter server. A general overview of the implementation is provided in Fig. 2. The Spark driver distributes the Resilient Distributed Dataset (RDD) of documents to different workers. Each worker pulls parts of the model from the parameter server and constructs corresponding alias tables. The worker then iterates over its local partition of the data and resamples the tokens using the Metropolis-Hastings algorithm. Updates are pushed asynchronously to the parameter server while the algorithm is running.

[2]https://github.com/rjagerman/glint/blob/master/src/main/scala/glint/models/client/async/PushFSM.scala

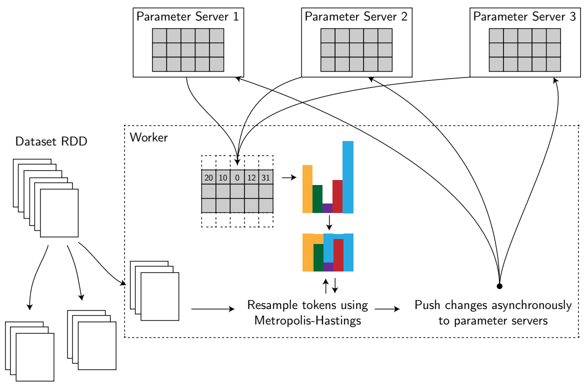
Parameter Server 1
Parameter Server 2
Parameter Server 3
Dataset RDD
Worker
20 10 0 12 31
Resample tokens using Metropolis-Hastings
Push changes asynchronously to parameter servers

Figure 2: Overview of the implementation. A dataset is split into different partitions by Spark. Tokens in each partition are resampled by the Metropolis-Hastings algorithm. Updates are pushed asynchronously to the parameter server.

4 EXPERIMENTS

There is no point in optimizing and scaling inference if the quality of the trained model should suffer. For this reason, we want to validate that the effectiveness of the trained model remains the same. It should be noted that our goal is **not** to outperform highly customized implementations such as LightLDA.

Instead, we aim to integrate state-of-the-art topic modeling with Spark such that large topic models can be efficiently computed in modern data processing pipelines. To this end, we compare our implementation against existing Spark implementations on the same hardware and configuration. We compare APS-LDA to two existing state-of-the-art LDA algorithms provided by Spark's MLLib: The EM algorithm [1] and the online algorithm [5]. We run our experiments on a compute cluster with 30 nodes, with a total of 480 CPU cores and 3.7TB RAM. The nodes are interconnected over 10Gb/s ethernet. The ClueWeb12 [7] corpus, a 27-terabyte Web crawl that contains 733 million Web documents, is used as the data set for our experiments.

To validate that our methods do not sacrifice the quality of the trained model we will compare the three algorithms on small subsets of ClueWeb12. We vary either the number of topics (20–80) or the size of the data set (50GB–200GB) to measure how the different systems scale with those variables and use perplexity as an indicator for topic model quality. Due to the large size of the data, a hyperparameter sweep is computationally prohibitively expensive and we set the LDA hyperparameters $\alpha = 0.05$ and $\beta = 0.001$ which we found to work well on the ClueWeb12 data set. We split the data in a 90% training set and a 10% test set and measure perplexity on the test set. Fig. 3 shows the results of the experiments. We observe that, barring some variations, the perplexity is roughly equal for all algorithms. However, our implementation has a significantly better runtime. We use a log-scale for the runtime in minutes.

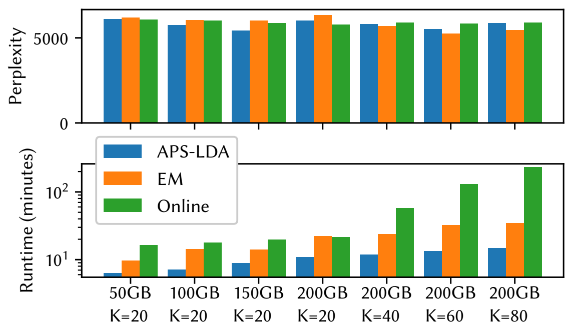

Figure 3: Performance of APS-LDA compared to the EM [1] and Online [5] algorithms for different data set sizes (50GB–200GB) and different numbers of topics *K* (20–80).

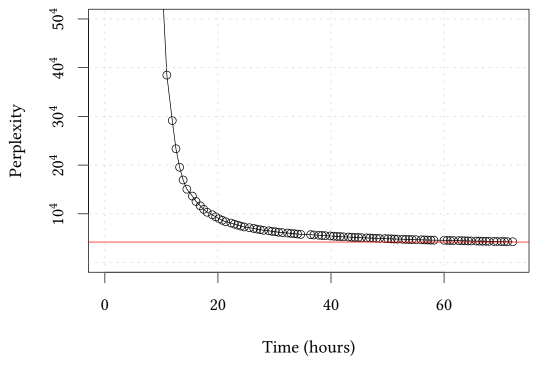

Figure 4: Perplexity of the 1,000-topic LDA model on ClueWeb12.

When attempting to increase the data set size beyond 200GB, the default Spark implementations cause numerous failures due to an increase in runtime and/or shuffle write size. Our implementation is able to effortlessly scale far beyond these limits and compute an LDA model on the full ClueWeb12 data set (27TB) with 1,000 topics in roughly 80 hours (see Fig. 4). This is an increase of nearly two orders of magnitude, both in terms of dataset size and number of topics, using identical hardware and configuration. We have made the final 1,000-topic LDA model publicly available in CSV format.[3]

5 CONCLUSION

We have presented APS-LDA, a distributed method for computing topic models on Web-scale data sets. It uses an asynchronous parameter server that is easily integrated with the cluster computing framework Spark. We conclude our work by revisiting the challenge that was presented in the introduction:

How do individual machines keep their model synchronized?

The asynchronous parameter server solves this by providing a distributed and concurrently accessed parameter space for the model being learned. The asynchronous design has several advantages over the traditional parameter server model: it prevents model staleness, makes mitigating network failure easier and enables the system to use more dynamic threading mechanisms.

Our proposed algorithm APS-LDA, is a thorough re-design of LightLDA that takes advantage of the asynchronous parameter server model. We have implemented this algorithm and the asynchronous parameter server using Spark, a popular cluster computing framework. The resulting architecture allows for the computation of topic models that are several orders of magnitude larger, in both dataset size and number of topics, than what was achievable using existing Spark implementations. The code of APS-LDA is available as open source (MIT licensed) and we are also sharing a 1,000-topic LDA model trained on ClueWeb 12.

Finally, there are two promising directions for future work: (1) Large-scale information retrieval tasks often require machine learning methods such as factorization machines and deep learning, which are known to benefit from the parameter server architecture [4]. By using an asynchronous parameter server, it may be possible to achieve significant speedups. (2) Our current implementation of the asynchronous parameter server uses a dense representation of the data, due to the garbage collection constraint imposed by the JVM runtime. By implementing sparse representations it is possible to scale even further as this will reduce both memory usage and network communication overhead.

Acknowledgments. This research was supported by Ahold Delhaize, Amsterdam Data Science, the Bloomberg Research Grant program, Criteo, the Dutch national program COMMIT, Elsevier, the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement nr 312827 (VOX-Pol), the Microsoft Research Ph.D. program, the Netherlands Institute for Sound and Vision, the Netherlands Organisation for Scientific Research (NWO) under project nrs 612.001.116, HOR-11-10, CI-14-25, 652.-002.001, 612.001.551, 652.001.003, and Yandex. All content represents the opinion of the authors, which is not necessarily shared or endorsed by their respective employers and/or sponsors.

REFERENCES

[1] A. Asuncion, M. Welling, P. Smyth, and Y. W. Teh. On smoothing and inference for topic models. In *UAI*, pages 27–34. AUAI, 2009.
[2] D. M. Blei, A. Y. Ng, and M. I. Jordan. Latent dirichlet allocation. *JMLR*, pages 993–1022, 2003.
[3] J. Chen, K. Li, J. Zhu, and W. Chen. WarpLDA: A simple and efficient O(1) algorithm for Latent Dirichlet Allocation. arXiv preprint arXiv:1510.08628, 2015.
[4] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, et al. Large scale distributed deep networks. In *NIPS*, pages 1223–1231, 2012.
[5] M. Hoffman, F. R. Bach, and D. M. Blei. Online learning for latent dirichlet allocation. In *NIPS*, pages 856–864, 2010.
[6] T. Hofmann. Probabilistic latent semantic indexing. In *SIGIR*, pages 50–57. ACM, 1999.
[7] Lemur Project. Clueweb12, 2016. [Online; http://lemurproject.org/clueweb12/; accessed 22-March-2016].
[8] M. Li, L. Zhou, Z. Yang, A. Li, F. Xia, D. G. Andersen, and A. Smola. Parameter server for distributed machine learning. In *NIPS*, page 2, 2013.
[9] M. Li, D. G. Andersen, J. W. Park, A. J. Smola, A. Ahmed, V. Josifovski, J. Long, E. J. Shekita, and B.-Y. Su. Scaling distributed machine learning with the parameter server. In *OSDI 14*, pages 583–598, 2014.
[10] P. Moritz, R. Nishihara, I. Stoica, and M. I. Jordan. SparkNet: Training deep networks in Spark. In *ICLR*, 2016.
[11] A. Welc, S. Jagannathan, and A. Hosking. Safe futures for java. *SIGPLAN*, pages 439–453, 2005.
[12] H.-F. Yu, C.-J. Hsieh, H. Yun, S. Vishwanathan, and I. S. Dhillon. A scalable asynchronous distributed algorithm for topic modeling. In *WWW*, pages 1340–1350. ACM, 2015.
[13] J. Yuan, F. Gao, Q. Ho, W. Dai, J. Wei, X. Zheng, E. P. Xing, T.-Y. Liu, and W.-Y. Ma. LightLDA: Big topic models on modest computer clusters. In *WWW*, pages 1351–1361, 2015.
[14] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, pages 2–2. USENIX, 2012.

[3]http://cake.da.inf.ethz.ch/clueweb-topicmodels/