F2M: Scalable Field-Aware Factorization Machines

Chao Ma, Yuze Liao, Yuan Wang, and Zhen Xiao*
Department of Computer Science
Peking University
{mahcao, lyz, xiaozhen}@net.pku.edu.cn

Abstract

Field-aware factorization machines (FFM) has experienced significant interest recently since its good performance on large-scale sparse data. Current systems for this model such as LibFFM, however, runs only on a single machine that would reach both the computation and storage limit when the data go very large. In this work, we introduce F2M, a distributed FFM implementation that can offer good performance and scalability. We will stand on a system designer's perspective to demonstrate how to pick the right machine learning and system techniques and bring them together, making the learning system efficient and scalable. The evaluation result proves the performance benefits of our system.

1 Introduction

As the widely used nonlinear models for recommendation and estimation, factorization machines (FM) [8] and its optimized version field-aware factorization machines (FFM) [11] have experienced significant interest in recently years. For recommender systems and polynomial generalized linear models, FFM offers a computationally efficient and powerful alternative, and it achieves excellent results by using low-rank expansion of higher degree polynomial terms. This makes the FFM a very attractive target for high-dimensional sparse data occurring in many settings such as computational advertising, user profiling, and recommendation. The impact of this model has been recognized in a number of machine learning and data mining challenges. In particular, almost all the wining solutions of the CTR prediction contest in kaggle used FFM [12, 13, 15].

Unfortunately, current systems for FFM such as LibFFM [9] runs only on a single machine, which could reach both the computation and storage limit when the data go very large. This occurs since each feature in this model need to be embedded into a *low-dimensional* $(f \times k)$ *matrix space*. A quick calculation shows that even on the modest datasets such as Criteo's CTR prediction contest [12] we have up to 10^7 features and 10^{11} model parameters (set $f = 100, k = 100$), requiring in the order of 100GB main memory. Worse yet, not only the memory cost but also the computation cost of the FFM is considerably larger than the conventional linear model. With the explosive growth of data on the Internet, the fact is that more and more FFM tasks are inadequate to be solved on a single machine and executed in a distributed manner has become a prerequisite for solving large-scale FFM problems.

In this paper we introduce a distributed FFM system called F2M, which can ensure both system efficiency and scalability. We propose a fast distributed optimization algorithm based on *asynchronous stochastic gradient descent*, along with *sparse regularization*. Moreover, F2M is based on an algorithm-specified parameter server (*PS*) framework [14, 17] with careful system design and optimizations. Our system is designed to minimize network traffic, maximize CPU and memory utilizations, provide cache-aware computation and support low-cost fault recovery mechanism. We evaluate F2M on real workloads and show that our system has the highly competitive performance and scalability. F2M is available under the Apache 2.0 license.

*The contact author is Zhen Xiao.

30th Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain.

Clicked ?	Gender	Occupation	Ad_Type
0	male	student	PC
1	female	cook	clothes
0	male	programmer	PC

(a)

Clicked ?	male	female	student	cook	programmer	PC	clothes
0	1	0	1	0	0	1	0
1	0	1	0	1	0	0	1
0	1	0	0	0	1	1	0

(b)

Figure 1: **Ad click-through rate (CTR) prediction.** Figure 1 (a) shows a CTR predication task with just three kinds of categorical feature. Figure 1 (b) shows the sparse feature vector after one-hot encoding.

Outline. The rest of this paper is structured as follows. In section 2 we begin with quick review of the conventional linear model, and then introduce FM and FFM. This is followed in section 3 by a description of the data-parallel machine learning. Section 4 has details about the design and optimization of the F2M system. The preliminary experimental results are provided in Section 5 and we conclude with a summary in Section 6.

2 From linear model to FM and FFM

Let us first consider a click-through rate (CTR) prediction task: predicting if individual users will click a given ad. As an example, suppose we just choose three kinds of categorical feature, as shown in *Figure 1*. Here we extracted sparse binary features via the widely-used one-hot encoding. We next discuss how to solve this problem by using linear mode, FM, as well as FFM.

Linear model. Generalized linear models (GLMs) have been widely used for large-scale datasets as its computational efficiency. The formulation of a conventional linear model is as follow:

$$f(X) = w_0 + \sum_{i=1}^{n} w_i x_i \tag{1}$$

As we all know, the capability of a linear model is limited when the training data is not linear separable. Generally, we can address this problem toward two directions. One is that we can use more powerful and nonlinear models such as kernel SVM, tree-based models like GBDT, as well as deep neural networks. Another general way to address this problem is to use feature combination such as the degree-2 (or higher degree) polynomial combination. Mathematically, the degree-2 polynomial combination has the following form:

$$f(X) = w_0 + \sum_{i=1}^{n} w_i x_i + \sum_{i=1}^{n} \sum_{j=i+1}^{n} w_{ij} x_i x_j \tag{2}$$

Intuitively, if the feature *'male'* and the feature *'programmer'* are combined, our model is more confident to predict that current user has a higher possibility to click the ad of PC. From formulation (2) we can also see that the degree-2 polynomial model has $O(n^2)$ parameters, where n is the parameter space of the linear model. By similar reasoning, a degree-m polynomial model has $O(n^m)$ parameters. Unfortunately, the typical setting in our real work is that of a very high-dimensional feature space n. As we mentioned before, we have up to 10^7 features on the Criteo's CTR dataset after one-hot encoding. As a result, even a quadratic model could be too expensive to estimate.

Factorization Machines. Rendle et al [8, 16] introduce a strategy called factorization machines (FM) for alleviating this problem via the low-rank expansion instead of a general high-dimensional expansion. Mathematically, the second-order FM has the following form:

$$f(X) = w_0 + \sum_{i=1}^{n} w_i x_i + \sum_{i=1}^{n} \sum_{j=i+1}^{n} (V_i \times V_j) x_i x_j \tag{3}$$

where, V_i is a low-dimensional vector embedded with feature i. From this formulation we can see that the FM uses a low-dimensional embedding of the (typically very sparse set of) features in x to a much smaller k-dimensional space. Compared to the high-degree polynomial model, the FM can

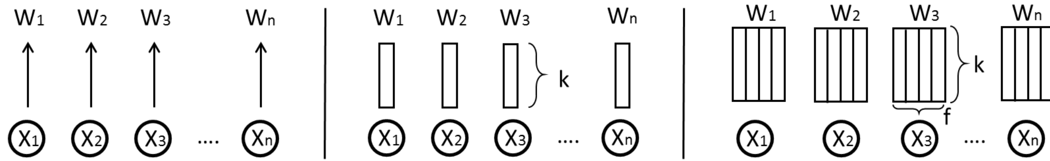

Figure 2: **Comparison of the conventional linear model, FM, and FFM.** Notice that we omit the bias term in linear model and omit the bias and linear term in FM and FFM.

reduce the number of parameters from a exponential level to a linear level $O(k \times n)$. Notice that the FM can also go beyond second-order. For example, the third-order FM has the following form:

$$f(X) = w_0 + \sum_{i=1}^{n} w_i x_i + \sum_{i<j} (V_i \times V_j) x_i x_j + \sum_{i<j<k} (V_i \times V_j \times V_k) x_i x_j x_k \tag{4}$$

Field-aware Factorization Machines. Unlike the FM, where each feature corresponds to an unique vector, the feature in FFM corresponds to a set of vectors (a low-dimensional $(f \times k)$ matrix). The number of the vectors for each feature equals to the number of the *field*. Recall the previous CTR prediction task, where we have three kinds of categorical feature but after one-hot encoding we extracted a sparse vector with seven binary features. So in this FFM task we can say that we have seven features but only have three fields. Mathematically, the FFM has the following form:

$$f(X) = w_0 + \sum_{i=1}^{n} w_i x_i + \sum_{i=1}^{n} \sum_{j=i+1}^{n} (V_{i,f_j} \times V_{j,f_i}) x_i x_j \tag{5}$$

where, V_{i,f_j} identifies the vector embedded with feature i when it combines with another feature that belongs to field j. Compared to the FM, FFM is more flexible and powerful but also has a larger parameter space $O(n \times f \times k)$. *Figure 2* gives an intuitive representation and summary of the linear model, FM, and FFM.

While the FFM can significantly reduce the number of parameters compared to the high-dimensional polynomial model, both the memory cost and the computation cost of the FFM is still considerably larger than the linear model. So the problem size remains formidable and we need tools for distributed inference: quite often the number of parameters significantly exceeds the amount of available memory on a single machine. This calls for distributed representations and optimization algorithms.

3 Data-parallel Machine Learning

In this section, we will talk about the data-parallel ML and focus on three questions: *What is the right abstraction for distributed ML systems? How do current systems support this abstraction? Why we choose the parameter server (PS) as our underlying computational engine?*

Observe that, with a few exceptions, almost all of the ML programs can be viewed as optimization-centric programs that adhere a general mathematical form: $min\ F(w)\ OR\ \ max\ F(w)$, where $F(w) = \sum_{i=1}^{n} \ell(x_i, y_i, w) + \Omega(w)$. We can typically resolve this problem by using the *gradient decent* that has the following form: $w_t = w_{t-1} - \eta * \Delta(w_{t-1},\ D)$, where, the most time-consuming step in this formulation is to compute the model update, i.e, the Δ function. Hence this computation process needs to be parallelized. For many machine learning algorithms, data-parallelism [18] is the most common parallelize scheme, where the Δ function can be parallelized over machines, each of which computes a sub-update on its own training data subset, following which the sub-updates are aggregated and applied to the model parameters. Mathematically, the data-parallel machine learning has the following form:

$$w_t = w_{t-1} - \eta * \sum_{p=1}^{P} \Delta(w_{t-1},\ D_p) \tag{6}$$

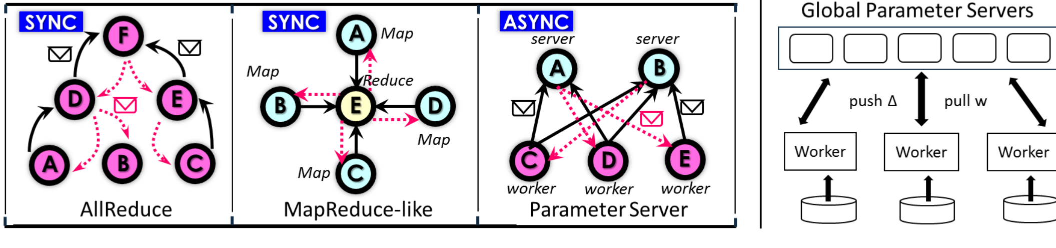

Figure 3: **Data-parallel machine learning architectures.** Figure 3 (left) demonstrates three different data-parallel architectures. Among these architectures, only the *PS* architecture can perform *asynchronous* and *fine-grained* computation. Figure 3 (right) shows a more detailed view of the *PS* architecture.

where, D_p is the data partition allocated to worker p. In each iteration, a subset data D_p is called a *mini-batch* in ML literature, and can be as small as one training data sample. In practice, the size of D_p requires manual or automatic tuning for the algorithms to work well.

Many existing systems naturally support data-parallel ML. In this section, we will provide a brief review and a comparative study of the most popular data-parallel ML systems. We will focus on their system architecture and programming model. *Figure 3* (left) summarizes three different data-parallel architectures.

AllReduce [10]. AllReduce is an MPI-primitive allowing normal sequential code to work in parallel implying very low programming overhead. In this model, each worker computes the sub-update Δ on its own data subset, and further these updates are aggregated via a *tree* [6], following which the aggregated result is broadcasted back to each worker. While the AllReduce interface provided by MPI is easy to use, its drawbacks are also obvious: First, this method waits for the slowest machine and therefore does not scale well to large shared clusters. Second, MPI implements this interface without fault-tolerance. Moreover, with the increasing size of ML models, both the networking bandwidth and memory cost could become the performance bottleneck.

MapReduce-like [19] (*data-flow*) systems such as Hadoop and Spark [20, 21] also simplified the task of building data-parallel ML applications. Based on them, the ML libraries such as Mahout [22] (based on Hadoop) and MLI [23] (based on Spark) have been widely used in both academia and industry. However, most of these systems adopt the *iterative MapReduce* paradigm that mandates *synchronous* and *coarse-grained* computation, which is similar to the AllReduce paradigm. Such inherent design for batch tasks often incurs great inefficiency and is inadequate to build the "big model" ML applications. Moreover, these systems typically require significant rewriting to existing single-machine ML code by using high-level APIs.

Parameter Server [23] system has recently emerged as an efficient approach to resolve the "big model" ML challenge. Under this model, both the training data and workloads are spread across worker nodes, while the server nodes maintain the globally shared parameters. In contrast to the *iterative MapReduce* model, computation in *PS* can be *asynchronous* and *fine-grained*, and hence can improve CPU utilization and reduce communication cost dramatically. Another benefit of the *PS* model is that *PS* does not require developers to change their programming habits on single-machine ML by using the *push* and *pull* interfaces. *Figure 3 (right)* shows the *PS* architecture.

Here we choose the *parameter server* as our underlying computational engine and the reasons are as follows: First, the model parameters of the FFM is typically very large, hence the coarse-grained and synchronous systems like Spark and MPI-AllReduce would face significant networking and synchronization overhead. Moreover, we can easily migrate the single-machine implementation of FFM to the distributed setting by using the *PS*. In this work, we designed an algorithm-specified parameter server with careful system design and optimizations, which will be discussed in the next section.

4 F2M: Design and Optimization

In this section, we will discuss the details about the design and optimization of F2M. In our presentation, we first cover the single-machine FFM implementation and then show how to seamlessly

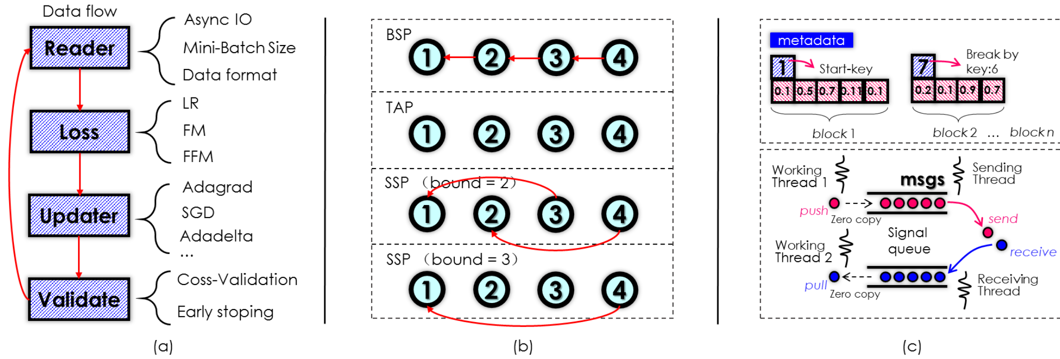

Figure 4: **The design and implementation of F2M.** Figure 4 (a) demonstrates our single-machine system design, where we divide our system into four modulars. Figure 4 (b) shows the different consistency models provided by F2M, including BSP, SSP and TAP. Figure 4 (c) shows the *batched block form* of message and the *signal queue*.

migrate it to a distributed setting. In addition, we will focus on the optimizations of network, which significantly improves our system performance. Notice that these optimizations we purpose in this section are general and can be easily adopted in other ML systems.

4.1 Single-machine Implementation

On the single machine, we focus on performance and flexibility when designing our system. As *Figure 4 (a)* shows, we divide F2M system into four modulars, each of which has the maximum flexibility to implement different capabilities and characteristics. In addition, to maximize the system perform, we use the *openmp* and *SSE3* to accelerate the training process.

4.2 Distributed Algorithm

On the distributed setting, we expend our single-machine implementation by using the asynchronous stochastic gradient descent under the *PS*. As we discussed before, our distributed implementation can easily retro-fit existing single-machine implementation by using the *push* and *pull* API provided by the *PS*, as shown below:

```
while(1) {pull(sub_mode); grad=CalGrad(data,sub_mode); push(grad);}
```

Distributed F2M has been divided into three modulars: the *master* node runs the control logic, *server* nodes update the model and the *worker* nodes compute the gradient. In a more detailed view, the server nodes store assigned parameters partition in its memory, and handles the aggregation and updating operations associated with that partition. Each worker node is responsible for storing a portion of the training data to compute local gradient. The master node maintains the bookkeeping of each worker and server process, which can be recovered without interrupting the computation when it crashes by non-catastrophic machine failures. Similar to existing systems, we assume that the master node failures are rare and hence provide no protection for that. Notice that worker nodes communicate only with the server nodes and the master node, not among themselves.

We also use the ℓ_1 regularization, which has been widely used in linear models for high dimensional data such as computational advertising [27]. We pick the penalty $\Omega[w] := \lambda_1 ||w||_1$, where λ_1 controls the degree of sparsity. The sparse model induced by the ℓ_1 regularization not only penalizes complex model, it also reduces the computation cost of the gradient and saves the communication traffic. It results in a smaller final model which further makes deploying this model on an online service easier.

In addition, F2M can support asynchronous computation and flexible consistency model, which has been shown to be more efficient than synchronous computation (e.g., BSP model [28]) and totally asynchronous computation [18] for many purposes [29], as shown in *Figure 4 (b)*.

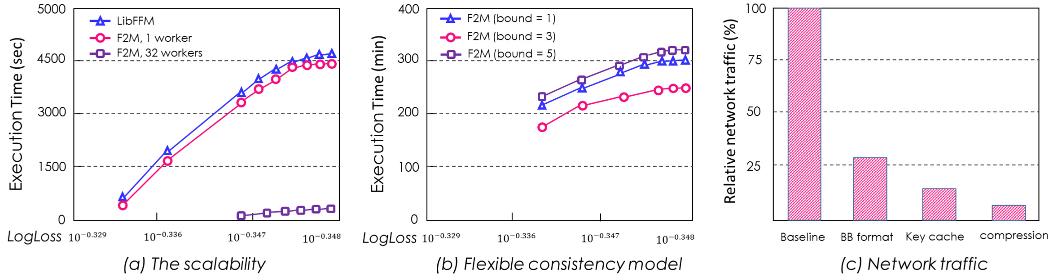

Figure 5: **Evaluation result.** Figure 5 (a) demonstrates the scalability of F2M. Using 32 machines, F2M can achieve 18.5 times speedup. F2M also has a highly competitive performance on single machine. Figure 5 (b) shows the effect on different consistency bound. Figure 5 (c) demonstrates the effect of our networking optimization, which can significantly reduce the network traffic and overhead.

4.3 System Optimization

Our system is developed by C++ and it re-uses a number of existing libraries such as MPICH2 for communication, Protobuf for serialization, and Snappy for data compression. F2M can run over the cluster resource manager such as Yarn [24] or Mesos [25], and also provides an easy way for deployment by using the docker container [26]. In this paper, we focus on the optimizations on communication, which significantly improves our system performance. We now focus on two aspects of our design, including the message compression and the message transport.

Message compression is desirable in large-scale ML problems, hence we use several approaches in our system to reduce the network traffic as much as possible: (1) We avoid sending single items and pack all of these items into a *batched block* form in each iteration, which reduces the size of each message, as shown in *Figure 4* (c). (2) Since many ML problems may use the same training data in different iterations, we cache the key lists in receive nodes. Later the senders can send only a hash value of this list rather than the list itself. (3) We use the Protobuf to serialize our message and further use the Snappy compression library to compress the serialized message. (4) We use lossy fixed-point compression for data communication. By default, both the model and gradient entries are represented as 32 bit floats. In F2M, we compress these values to lower precision integers.

Message transport in F2M also has been designed carefully. To maximize CPU utilization and reduce latency, F2M implements two signal queues as in-memory buffer, as shown in *Figure 4* (c). As this figure shows, we separate the computation and networking in different threads, hence the networking threads will not block the computation. In addition, zero copy technique is also used in signal queue, where we pass pointers rather than copy data values to the queue.

5 Preliminary Evaluation

Our evaluation is based on a cluster of 32 machines, each of which has a 16 cores Intel Xeon E5620 (2.40 GHz) processor with 32G memory. All machines are connected via 1Gb Ethernet. Our datasets come from the Cretio CTR [37]. The former is used in a recent Kaggle competition, for which we used the first 80% for training and the rest for test. The highlights of our results can be found in *Figure 5*. If this paper is accepted, we plan to demo more detailed evaluation of our system during the workshop.

6 Conclusion and Future Work

In this work, we demonstrate a distributed FFM implementation called F2M, which can offer good performance and scalability. We propose a fast distributed optimization algorithm based on asynchronous stochastic gradient descent. Moreover, F2M is based on an algorithm-specified parameter server framework with careful system design and optimizations. The evaluation result proves the performance benefits of our system.

In the future, we plan to support a wide range of front ends for languages like Python, R, Go and perhaps others, making our system easier to use.

7 Acknowledgments

The authors would like to thank the anonymous reviewers for their comments. This work was supported by the National Grand Fundamental Research 973 Program of China under Grant No.2014CB340405 and the National Natural Science Foundation of China under Grant No.61572044. The contact author is Zhen Xiao.

References

[1] POWER, R., AND LI, J. Piccolo: Building fast, distributed programs with partitioned tables. In OSDI (2010).

[2] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: distributed data-parallel programs from sequential building blocks. In ACM SIGOPS Operating Systems Review (2007).

[3] LI, H., KADAV, A., KRUUS, E., AND UNGUREANU, C. Malt: distributed data-parallelism for existing ml applications. In Proceedings of the Tenth European Conference on Computer Systems (2015).

[4] Chien-Chin Huang, Qi Chen, Zhaoguo Wang, Russell Power, Jorge Ortiz, Jinyang Li, and Zhen Xiao. Spartan: A Distributed Array Framework with Smart Tiling Proc. of the USENIX Annual Technical Conference (ATC 2015), July 2015.

[5] Chao Ma, Yan Ni, and Zhen Xiao. Brook: An Easy and Efficient Framework for Distributed Machine Learning Proc. of the LearningSys Workshop on NIPS 2015, December 2015.

[6] AGARWAL, A., CHAPELLE, O., DUDt'IK, M., AND LANGFORD, J. A reliable effective terascale linear learning system. The Journal of Machine Learning Research (2014).

[7] Hasselmo, M.E., Schnell, E. & Barkai, E. (1995) Dynamics of learning and recall at excitatory recurrent synapses and cholinergic modulation in rat hippocampal region CA3. *Journal of Neuroscience* **15**(7):5249-5262.

[8] S. Rendle and L. Schmidt-Thieme. Pairwise interaction tensor factorization for personalized tag recommendation. In Web search and data mining, pages 8190. ACM, 2010.

[9] LibFFM: https://github.com/guestwalk/libffm

[10] PATARASUK, P., AND YUAN, X. Bandwidth efficient all-reduce operation on tree topologies. In Parallel and Distributed Processing Symposium, 2007. IPDPS 2007. IEEE International (2007).

[11] FFM: http://www.csie.ntu.edu.tw/ r01922136/slides/ffm.pdf

[12] Display Advertising Challenge: https://www.kaggle.com/c/criteo-display-ad-challenge

[13] Avito Context Ad Clicks: https://www.kaggle.com/c/avito-context-ad-clicks

[14] XING, E. P., HO, Q., DAI, W., KIM, J. K., WEI, J., LEE, S., ZHENG, X., XIE, P., KUMAR, A., AND YU, Y. Petuum: A new platform for distributed machine

[15] Avazu Click-Through Rate Prediction: https://www.kaggle.com/c/avazu-ctr-prediction

[16] S. Rendle. Time-Variant Factorization Models ContextAware Ranking with Factorization Models. volume 330 of Studies in Computational Intelligence, chapter 9, pages 137 153. 2011. ISBN 978-3-642-16897-0.

[17] LI, M., ANDERSEN, D. G., PARK, J. W., SMOLA, A. J., AHMED, A., JOS IFOVSKI, V., LONG, J., SHEKITA, E. J., AND SU, B.-Y. Scaling distributed machine learning with the parameter server. In 11th USENIX Symposium on Operating System

[18] DEAN, J., CORRADO, G., MONGA, R., CHEN, K., DEVIN, M., MAO, M., SENIOR, A., TUCKER, P., YANG, K., LE, Q. V., ET AL. Large scale distributed deep networks. In Advances in Neural Information Processing Systems (2012), pp. 12231231.

[19] DEAN, J., AND GHEMAWAT, S. Mapreduce: simplified data processing on large clusters. Communications of the ACM (2008).

[20] WHITE, T. Hadoop: The definitive guide. OReilly Media, Inc., 2012.

[21] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M. J., SHENKER, S., AND STOICA, I. Spark: Cluster computing with working sets. HotCloud 10 (2010).

[22] OWEN, S., ANIL, R., DUNNING, T., AND FRIEDMAN, E. Mahout in action. Manning Shelter Island, 2011.

[23] SPARKS, E. R., TALWALKAR, A., SMITH, V., KOTTALAM, J., PAN, X., GONZALEZ, J., FRANKLIN, M. J., JORDAN, M. I., AND KRASKA, T. Mli: An api for distributed machine learning. In Data Mining (ICDM), 2013 IEEE 13th International Conference on (2013), IEEE, pp. 11871192.

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

[25] Hindman, Benjamin, et al. Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center. NSDI. Vol. 11. 2011.

[26] Docker: http://www.docker.com

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

[28] MCCOLL, W. F. Scalability, portability and predictability: The bsp approach to parallel programming. Future Generation Computer Systems (1996).

[29] CUI, H., CIPAR, J., HO, Q., KIM, J. K., LEE, S., KUMAR, A., WEI, J., DAI, W., GANGER, G. R., GIBBONS, P. B., ET AL. Exploiting bounded staleness to speed up big data analytics. In 2014 USENIX Annual Technical Conference (USENIX ATC 14) (2014).