# Distributed Matrix Factorization with MapReduce using a series of Broadcast-Joins

Sebastian Schelter Christoph Boden Martin Schenck
Alexander Alexandrov Volker Markl

Technische Universität Berlin, Germany
firstname.lastname@tu-berlin.de

## ABSTRACT

The efficient, distributed factorization of large matrices on clusters of commodity machines is crucial to applying latent factor models in industrial-scale recommender systems. We propose an efficient, data-parallel low-rank matrix factorization with Alternating Least Squares which uses a series of broadcast-joins that can be efficiently executed with MapReduce.

We empirically show that the performance of our solution is suitable for real-world use cases. We present experiments on two publicly available datasets and on a synthetic dataset termed *Bigflix*, generated from the Netflix dataset. Bigflix contains 25 million users and more than 5 billion ratings, mimicking data sizes recently reported as Netflix' production workload. We demonstrate that our approach is able to run an iteration of Alternating Least Squares in six minutes on this dataset. Our implementation has been contributed to the open source machine learning library Apache Mahout.

## Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

## Keywords

Scalable Collaborative Filtering, MapReduce

## 1. INTRODUCTION

Today's information overload has triggered the development of recommender systems: intelligent filters that learn about the users' preferences and find the information most relevant to them.

On the technical side, the processing efficiency and scalability of the systems becomes a major concern in light of rapidly growing data sizes [2]. In a production environment, the offline computations necessary to run a recommender system must be periodically executed as part of larger analytical workflows and thus have to adhere to strict time and resource constraints. Additionally, computations must be executed repeatedly and in parallel to find good model parameters via cross-validation. For economic and operational reasons it is often undesirable to execute such offline computations on a single machine: this machine might fail and with growing data sizes, constant hardware upgrades might be necessary to improve the machine's performance to meet the time constraints. Due to these disadvantages, a single machine solution can quickly become expensive and hard to operate. Running data-intensive, analytical computations in a parallel and fault-tolerant manner on a cluster of commodity machines [9, 14] poses a solution to this problem. The computation becomes independent of single machine failures and performance can be increased by simply adding more machines to the cluster.

In recent research, we showed how to efficiently apply neighborhood methods in such a scenario [18]. When we apply advanced techniques such as latent factor models [13] in a distributed manner, we face a problematic situation: The algorithms used to solve latent factor models are of iterative nature. The execution of iterative programs imposes serious overhead when carried out with established paradigms such as MapReduce [9, 14], where every iteration is scheduled as a separate job and has to re-scan all iteration-invariant data. Although recently proposed specialized systems [15, 22] provide superior performance for iterative algorithms, they have the drawback that they are difficult to employ in production settings, where the existing analytics infrastructure is very often built on top of the open source MapReduce implementation Hadoop [3] and its ecosystem. Integrating specialized systems into such an environment requires heterogeneous analytics pipelines where e.g. preprocessing is done by a MapReduce system and model training is conducted by a specialized machine learning system. Such pipelines are undesirable because they are complex to setup and have high maintenance cost [14].

In this paper, we show that the low-rank matrix factorization of commonly used datasets in recommendation mining is a corner case that can be efficiently computed with MapReduce although the underlying computation is iterative. The performance of our proposed solution is suitable for production settings and at the same time, our approach is easy to integrate into existing analytics infrastructures, as it runs on Hadoop. We describe how to execute the underlying computation using only a series of broadcast-joins [7], which are efficiently executable in MapReduce and avoid a lot of the common drawbacks of the Hadoop framework.

For evaluation, we conduct experiments on two publicly available datasets consisting of several hundred million movie and song ratings. To test our approach on industrial-scale, we run experiments on a synthetic dataset with 25 million users and more than 5 billion ratings, mimicking real-world data sizes recently reported by Netflix [2].

The contributions of this paper are the following: (1) We show how to efficiently execute an iterative low-rank matrix factorization algorithm with MapReduce in a scalable manner. (2) We provide

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*RecSys'13,* October 12–16, 2013, Hong Kong, China.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-2409-0/13/10 ...$15.00.
http://dx.doi.org/10.1145/2507157.2507195.

an experimental evaluation on two real-world datasets and on a synthetic industrial-scale dataset with more than 5 billion ratings.

The rest of the paper is organized as follows. Section 2 briefly introduces MapReduce and latent factor models. Section 3 in detail discusses our proposed approach. Section 4 describes related work. Section 5 gives the results we obtain from our experimental evaluation, whereas Section 6 concludes and discusses future work.

## 2. PRELIMINARIES

### 2.1 MapReduce and its limitations

MapReduce [9] is a functional paradigm for data-intensive parallel processing on shared-nothing clusters running a distributed filesystem (DFS). Under this paradigm, the user has to express algorithms as first-order functions supplied to the second-order functions *map* and *reduce*. The framework then automatically parallelizes the program and takes care of details such as scheduling the program's execution on the cluster, managing the inter-machine communication as well as coping with machine failures. Hadoop [3] is a popular and widely deployed open source implementation of MapReduce. In Hadoop, jobs are executed as a pipeline *map-shuffle-reduce*, where the map function is invoked on the input data in the DFS in parallel, the output tuples are grouped by their key and then sent to the reducer machines in the shuffle phase. The receiving machines merge the tuples, invoke the reduce function on all tuples sharing the same key and finally write the output to the DFS . The shuffle phase is typically the most costly operation as the mappers' output is spilled to disk at first and each reducer downloads its assigned data from every mapper afterwards.

In general, such a framework is a poor fit for iterative algorithms, as each iteration has to be scheduled as a single MapReduce job with a high startup cost (potentially up to tens of seconds). Further, the system creates a lot of unnecessary I/O and network traffic as all static, iteration-invariant data has to be re-read from disk and re-processed during each iteration and the intermediary result of each iteration has to be materialized in the DFS.

### 2.2 Collaborative Filtering

Collaborative Filtering (CF) is a popular approach in recommender systems which analyzes the historical interactions between a user and an arbitrary kind of item. Based on patterns found in the historical data, a CF algorithm recommends new, potentially high-preferred items to the users. Let $A$ be a $|C| \times |P|$ matrix holding all known interactions between a set of users $C$ and a set of items $P$. If a user $i$ interacted with an item $j$, then $a_{ij}$ holds a numeric value representing the strength of the interaction.

During the much-noticed Netflix prize [6], "latent factor models", approaches to CF that leverage a low-rank matrix factorization of the interaction data, became very popular [13]. The idea is to approximately factor the sparse $|C| \times |P|$ matrix $A$ into the product of two rank $r$ feature matrices $U$ and $M$ such that $A \approx UM$. The $|C| \times r$ matrix $U$ models the latent features of the users, (the rows of $A$), while the $r \times |P|$ matrix $M$ models the latent features of the items (the columns of $A$). A prediction for the strength of the relation between a user and an item (e.g., the preference of a user towards a movie) is given by the dot product $u_i^\top m_j$ of the vectors for user $i$ and item $j$ in the low-dimensional feature space. A popular technique to compute such a factorization is Stochastic Gradient Descent (SGD) [11,13,20], which randomly loops through all observed interactions $a_{ij}$, computes the error of the prediction $u_i^\top m_j$ for each interaction and modifies the model parameters in the opposite direction of the gradient. Another technique is Alternating Least Squares (ALS) [12,23], which repeatedly keeps one of the unknown matrices (either $U$ or $M$) fixed, so that the other one can be optimally re-computed. ALS then rotates between re-computing the rows of $U$ in one step and the columns of $M$ in the subsequent step. Note that such alternating convex optimization techniques can also be applied to CF approaches that learn a ranking function [21].

## 3. APPROACH

### 3.1 Parallelization

For single machine implementations, SGD is the preferred technique to compute a low-rank matrix factorization [4,10]. SGD is easy to implement and computationally less expensive than ALS, where every iteration runs in $O((|C| + |P|) * r^3)$, as $|C| + |P|$ linear systems have to be solved. Unfortunately, SGD is inherently sequential, because it updates the model parameters after each processed interaction. Techniques for parallel SGD have been proposed, yet they are either hard to implement, exhibit slow convergence or require shared-memory [11,16,20].

Following this rationale, we chose ALS for our parallel implementation. Although it is computationally more expensive than SGD, it naturally amends itself to parallelization. When we re-compute the user feature matrix $U$ for example, $u_i$, the $i$-th row of $U$, can be re-computed by solving a least squares problem only including $a_i$, the $i$-th row of $A$, which holds user $i$'s interactions, and all the columns $m_j$ of $M$ that correspond to non-zero entries in $a_i$. This re-computation of $u_i$ is independent from the re-computation of all other rows of $U$ and therefore, the re-computation of $U$ is easy to parallelize if we manage to guarantee efficient data access to the rows of $A$ and the corresponding columns from $M$. In the following we refer to the sequence of re-computing of $U$ followed by re-computing $M$ as a single iteration in ALS.

From a data processing perspective, this means that we have to conduct a parallel join between the interaction data $A$ and $M$ (the item features) in order to re-compute the rows of $U$. Analogously, we have to conduct a parallel join between $A$ and $U$ (the user features) to re-compute $M$. Finding an efficient execution strategy for these joins is crucial to the performance of our proposed parallel solution.

### 3.2 Execution with MapReduce

When executing joins in a shared-nothing environment, minimizing the required amount of inter-machine communication is decisive for the performance of the execution, as network bandwidth is the most scarce resource in a cluster. For matrix factorization with ALS, we have to alternatingly join $A$ with $M$ and $U$. In both cases, the interaction matrix $A$ is usually much larger than any of the feature matrices. We limit our approach to use-cases where neither $U$ nor $M$ need to be partitioned (which means they individually fit into the memory of a single machine of the cluster). A rough estimate of the required memory for the re-computation steps in ALS is $\max(|C|, |P|) * r * 8$ byte, as alternatingly, a single dense double-precision representation of the matrices $U$ or $M$ has to be stored in memory on each machine. Even for 10 million users or items and a rank of 100, the estimated required memory would be less than 8 gigabyte, which can easily be handled by today's commodity hardware. Our experiments in Section 5 show that, despite this limitation, we can handle datasets with billions of data points.

In such a setting, an efficient way to implement the necessary joins for ALS in MapReduce is to use a parallel broadcast-join [7]. The smaller dataset ($U$ or $M$) is replicated to every machine of the cluster. Each machine already holds a local partition of $A$ which is stored in the DFS. Then the join between the local partition of $A$ and the replicated copy of $M$ (and analogously between the local partition of $A$ and $U$) can be executed by a map operator. This operator can additionally implement the logic to re-compute the feature vectors from the join result, which means that we can execute a whole re-computation of $U$ or $M$ with a single map operator.

Figure 1 exemplarily illustrates the parallel join for re-computing $U$ using three machines. We broadcast $M$ to all participating machines, which create a hashtable for its contents, the item feature vectors. $A$ is stored in the DFS partitioned by its rows and forms the input for the map operator (cf. Figure 1, where e.g., $A(1)$ refers to partition 1 of $A$). The map operator reads a row $a_i$ of $A$ (the interaction history of user $i$) and selects all the item feature vectors $m_j$ from the hashtable holding $M$ that correspond to non-zero entries $j$ in $a_i$. Next, the map operator solves a linear system created from the interactions and item feature vectors and writes back its result, the re-computed feature vector $u_i$ for user $i$. The re-computation of $M$ works analogously, with the only difference that we need to broadcast $U$ and store $A$ partitioned by its columns (the interactions per item) in the DFS.

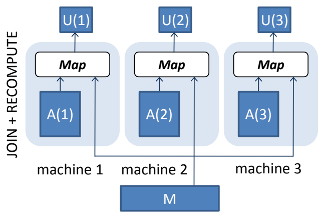

**Figure 1: Parallel re-computation of user features by a broadcast-join.**

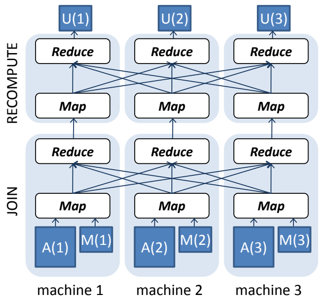

**Figure 2: Parallel re-computation of user features by a repartition-join.**

The proposed approach is able to avoid some of the drawbacks of MapReduce and the Hadoop implementation described in Section 2.1. It uses only map jobs that are easier to schedule than jobs containing map and reduce operators. Additionally, the costly shuffle-phase is avoided, in which all data would be sorted and sent over the network. As we can execute the join and the re-computation using a single job, we also spare to materialize the join result in the DFS. Our implementation contains multithreaded mappers that leverage all cores of the worker machines for the re-computation of the feature matrices and uses JBlas [5] for solving the dense linear systems present in ALS. The broadcast of the feature matrix is conducted via Hadoop's distributed cache in the initialization phase of each re-computation. Furthermore, we configure Hadoop to reuse the VMs on the worker machines and cache the feature matrices in memory to avoid that later scheduled mappers have to reread the data. The main drawback of a broadcast approach is that every additional machine in the cluster requires another copy of the feature matrix to be sent over the network. An alternative would be to use a repartition join [7] with constant network traffic and linear scale-out (cf., Figure 2). However, this technique has an enormous constant overhead. Let $n_i$ denote the number of interactions, $n_m$ the number of machines in the cluster, $n_r$ the replication factor of the DFS and $n_e$ the number of users or items[1]. If we employ a repartition-join we need two map-reduce jobs per re-computation then. In the first job, we conduct the join which sends the interaction matrix $A$ and either $U$ or $M$ over the network, accounting to $n_i * 4 + n_e * r * 8$ bytes of network traffic. The result must be materialized in the DFS, which requires another $(n_i * 4 + n_e * r * 8) * n_r$ bytes of traffic. The re-computation of the feature matrix must be conducted via a second map-reduce job that sends all the ratings and corresponding feature vectors per user or item over the network, accounting for an additional $n_i * r * 8$ bytes. On the contrary, the traffic for our proposed broadcast-join technique depends on the number of machines and accounts to $n_e * r * n_m * 8$ bytes. Applying these estimations to the datasets used for our experiments, the cluster size would have to exceed several hundred machines to have our broadcast-join technique cause more network traffic than a computation via repartition-join. Further, this argumentation only looks at the required network traffic and does not account for the fact that the computation via repartition-join would also need to sort and materialize the intermediate data in each step. We conclude that our approach with a series of broadcast-joins is to be preferred for common production scenarios.

[1]we assume that rating values are stored with single precision

## 4. RELATED WORK

Zhou et. al. describe a distributed implementation using Matlab, which is only suitable for running scientific experiments as it offers no fault tolerance [23]. GraphLab [15] is a specialized system for parallel machine learning, where programs operate on a graph expressing the computational dependencies of the data. It provides an asynchronous implementation of ALS. Gemulla et. al presented a parallel matrix factorization using Stochastic Gradient Descent [11], which leverages an intelligent partitioning to avoid conflicting updates. They switched their implementation to MPI [20] due to the overhead incurred by Hadoop. SystemML [19] is a proprietary library for machine learning on MapReduce which allows the declarative specification of matrix factorization algorithms. The recommender system of Google News [8] uses a MapReduce implementation of probabilistic latent semantic indexing. As Google uses a proprietary implementation of MapReduce, it is hard to assess the technical details of this approach.

## 5. EVALUATION

In this section, we experimentally show that our approach is suitable for real-world production settings where an approach has to be compatible to the Hadoop ecosystem and has to run in a few hours to be integratable into analytical workflows. We solely focus on measuring the runtime of computing a factorization, as the prediction quality of the ALS approach is well studied [11, 12, 23]. We use a formulation of ALS that is aimed at rating prediction [23].

*Setup:* The cluster for our experiments consists of 26 machines running Java 7 and Hadoop 1.0.4 [3]. Each machine has two 8-core Opteron CPUs, 32 GB memory and four 1 TB disk drives.

*Datasets:*[2] We use two publicly available datasets for our experiments: A set of 100,480,507 ratings given to 17,770 movies by 480,189 users from the Netflix prize [6] and another set comprised of 717,872,016 ratings given to 136,736 songs by 1,823,179 users of the Yahoo! Music community[3]. For tests at industrial-scale, we generate a synthetic dataset termed Bigflix using the Myriad data generation toolkit [1]. From the training set of the Netflix prize, we extract the probability of rating each item and increase the item space by a factor of six to gain more than 100,000 movies. Next, we extract the distribution of ratings per user. We configure Myriad to create data for 25 million users by the following process: For each user, Myriad samples a corresponding number of ratings from the extracted distribution of ratings per user. Then, Myriad samples

[2]Details about our experiments can be found at http://goo.gl/YXj9B
[3]http://webscope.sandbox.yahoo.com/catalog.php?datatype=r

an item from the item probability distribution for each rating. The corresponding rating value is simply chosen from a uniform distribution, as we do not interpret the result anyways, but only want to look at the performance of computing the factorization. Bigflix contains 5,231,536,647 interactions and mimicks the 25 million subscribers and 5 billion ratings, which Netflix recently reported as its current production workload [2].

*Experimental results:* We start by measuring the average runtime per individual re-computation of $U$ and $M$ for different feature space sizes on the Netflix dataset and on the Yahoo Songs dataset (cf., Figure 3) using 26 machines. For Netflix, the average runtime per re-computation always lies between 35 and 60 seconds regardless of the feature space size. This is a clear indication that for this small dataset the runtime is dominated by the scheduling overhead of Hadoop. For the larger Yahoo Songs dataset we see the same behavior for small feature space sizes and observe that the computation becomes dominated by the time to broadcast one feature matrix and re-compute the other one for larger feature space sizes of 50 and 100. We notice that re-computing $M$ is much more costly then re-computing $U$. This happens because in this dataset, the number of users is nearly twenty times as large as the number of items, which means that it takes much longer to broadcast $U$. The experiments show that we can run 37 to 47 iterations per hour on Netflix and 15 to 38 iterations per hour on the Yahoo Songs dataset (e.g., ALS typically needs about 15 iterations to converge on Netflix [23]). This shows that our approach is easily able to compute factorizations of datasets with hundreds of millions of interactions repeatedly per day.

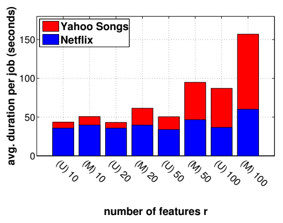

**Figure 3: Runtimes on Netflix and Yahoo Songs with different feature space sizes.**

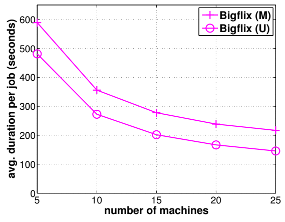

**Figure 4: Runtimes on Bigflix with different cluster sizes.**

Finally we run scale-out tests on Bigflix: we measure the average runtime per re-computation of $U$ and $M$ during five iterations of ALS on this dataset on clusters of 5, 10, 15, 20 and 25 machines (cf., Figure 4) conducting a factorization of rank ten. With 5 machines, an iteration takes about 19 minutes and with 25 machines, we are able to bring this down to 6 minutes. We observe that the computation speedup does not linearly scale with the number of machines. This behavior is expected because of the broadcast-join we employ, where every additional machine causes another copy of the feature matrix to be sent over the network. As discussed in Section 3.2, on clusters with less than several hundred machines, our chosen approach is much more performant than a repartition-join, although the latter is linearly scaling. With 25 machines, we can run about 9 to 10 iterations per hour, which allows us to obtain a factorization in a few hours, a timeframe completely suitable for a production setting.

## 6. SUMMARY

We presented an efficient scalable approach for a data-parallel low-rank matrix factorization using Alternating Least Squares on a cluster of commodity machines. We explained our choice of ALS as algorithm and described how to implement it on MapReduce with a series of broadcast-joins, which can be efficiently executed using only map jobs. We experimentally validated our approach on two large datasets commonly used in research and on a synthetic dataset with more than 5 billion datapoints, which mimicks an industry usecase. Furthermore, our code has partially been contributed to the open source machine learning library Apache Mahout [4]. In future work we would like to compare our approach to implementations in specialized systems [15, 22] and see if we can incorporate approximate solutions for ALS [17].

## 7. ACKNOWLEDGMENTS

We thank Zeno Gantner for early comments on the paper. The research is funded by the European Union (EU) under grants no. 25789 'ROBUST' and no. 270137 'SCAPE', the BMBF under grant no. 01ISI2033 'RADAR' and the European Institute of Innovation and Technology (EIT). We used data provided by 'Yahoo! Academic Relations'.

## 8. REFERENCES

[1] A. Alexandrov, K. Tzoumas, and V. Markl. Myriad: scalable and expressive data generation. *VLDB*, 2012.

[2] X. Amatriain. Building industrial-scale real-world recommender systems. *RecSys*, 2012, http://goo.gl/cqKAi.

[3] Apache Hadoop, http://hadoop.apache.org.

[4] Apache Mahout, http://mahout.apache.org.

[5] JBlas, http://jblas.org.

[6] J. Bennett and S. Lanning. The netflix prize. 2007.

[7] S. Blanas, J. M. Patel, V. Ercegovac, J. Rao, E. J. Shekita, and Y. Tian. A comparison of join algorithms for log processing in mapreduce. *SIGMOD*, 2010.

[8] A. S. Das, M. Datar, A. Garg, and S. Rajaram. Google news personalization: scalable online collaborative filtering. *WWW*, 2007.

[9] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 2008.

[10] Z. Gantner, S. Rendle, C. Freudenthaler, and L. Schmidt-Thieme. Mymedialite: a free recommender system library. *RecSys*, 2011.

[11] R. Gemulla, E. Nijkamp, P. J. Haas, and Y. Sismanis. Large-scale matrix factorization with distributed stochastic gradient descent. *KDD*, 2011.

[12] Y. Hu, Y. Koren, and C. Volinsky. Collaborative filtering for implicit feedback datasets. *ICDM*, 2008.

[13] Y. Koren, R. Bell, and C. Volinsky. Matrix factorization techniques for recommender systems. *Computer*, 2009.

[14] J. Lin. MapReduce is Good Enough? If All You Have is a Hammer, Throw Away Everything That's Not a Nail! *CoRR*, 2012

[15] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. Hellerstein. Distributed graphlab: A framework for machine learning in the cloud. *PVLDB*, 2012.

[16] F. Niu, B. Recht, C. Ré, and S. Wright. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. *NIPS*, 2011.

[17] I. Pilászy, D. Zibriczky and D. Tikk. Fast als-based matrix factorization for explicit and implicit feedback datasets. *RecSys*, 2010.

[18] S. Schelter, C. Boden, and V. Markl. Scalable Similarity-Based Neighborhood Methods with MapReduce. *RecSys*, 2012.

[19] A. Ghoting, R. Krishnamurthy, E. Pednault, B. Reinwald, V. Sindhwani, S. Tatikonda, T. Yuanyuan and S. Vaithyanathan. SystemML: Declarative machine learning on MapReduce. *ICDE*, 2011.

[20] C. Teflioudi, F. Makari, and R. Gemulla. Distributed matrix completion. *ICDM*, 2012.

[21] M. Weimer, A. Karatzoglou, Q. Viet Le and A. Smola. CofiRank - Maximum Margin Matrix Factorization for Collaborative Ranking. *NIPS*, 2007.

[22] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: cluster computing with working sets. *HotCloud*, 2010.

[23] Y. Zhou, D. Wilkinson, R. Schreiber, and R. Pan. Large-scale parallel collaborative filtering for the netflix prize. *AAIM*, 2008.