Collaborative filtering recommendation algorithm based on Hadoop and Spark

Article · June 2015

DOI: 10.1109/ICIT.2015.7125310

CITATIONS
11

READS
299

2 authors, including:

Olgierd Unold
Wroclaw University of Science and Technology
93 PUBLICATIONS **283** CITATIONS

SEE PROFILE

Collaborative Filtering Recommendation Algorithm based on Hadoop and Spark

Bartosz Kupisz
Chair of Computer Engineering
Wroclaw University of Technology
Wyb. Wyspianskiego 27, 50-370 Wroclaw, Poland
Email: kupisz.bartosz@gmail.com

Olgierd Unold
Chair of Computer Engineering
Wroclaw University of Technology
Wyb. Wyspianskiego 27, 50-370 Wroclaw, Poland
Email: olgierd.unold@pwr.edu.pl

***Abstract*—The aim of this work was to develop and compare recommendation systems which use the item-based collaborative filtering algorithm, based on Hadoop and Spark. Data for the research were gathered from a real social portal the users of which can express their preferences regarding the applications on offer. The Hadoop version was implemented with the use of the Mahout library which was an element of the Hadoop ecosystem. The authors original solution was implemented with the use of the Apache Spark platform and the Scala programming language. The applied similarity measure was the Tanimoto coefficient which provides the most precise results for the available data. The initial assumptions were confirmed as the solution based on the Apache Spark platform turned out to be more efficient.**

I. Introduction

Recommendation systems are the most recognizable and currently the most widely used technique of machine learning [1]. They have an undeniable impact on the personalization of content on the Internet, so prevalent nowadays. There is great interest in them, especially in e-Commerce, as their implementation raises sales by an estimate of 8 to 10 percent. During the last few years, as a huge amount of data is used for generating recommendations, those systems are equipped with solutions characteristic of the problems of Big Data.

The task of recommendation systems is to present users with information about the items in which they might be interested. The systems differ from one another in the way they analyze the available data which, in turn, make it possible to evaluate the degree to which users like particular products. The techniques are used in personalized recommendation systems can be classified into three basic groups: content-based, those based on collaborative filtering (abbreviated to CF), and hybrid systems [2].

This work focuses on systems which use collaborative filtering, because of their universality. Note that in a CF algorithm, it is expensive to compute the similarity of users (or items) as the algorithm is required to search entire database to find the potential neighbors for a target user (item). It requires computation that increase linearly with the growth of both the number of users and items (a CF algorithm has a worst case complexity of $O(UI)$, where U is the number of users and I is the number of items). Therefore, many algorithms are either slowed down or require additional resources such as computation power or memory. This is so-called the scalability problems, which is one of the key challenge for CFs [3].

Recently published works proved that it is possible to parallelize collaborative filtering algorithms with Hadoop technology [4], [5], which is dedicated to solving complex and distributable problems. Admittedly, however, this approach based on MapReduce paradigm has not favorable scalability and computation-cost efficiency if the data size increase.

This paper presents a new solution to item-based CF based on the Apache Spark platform - a new engine for large-scale data processing. We develop and compare item-based collaborative filtering algorithm using two cluster computing frameworks: Hadoop's disk-based MapReduce paradigm and Spark's in-memory based RDD paradigm. The implementation of the CF algorithm based on Hadoop platform was done using the Mahout library [6], [7]. Paralleled item-based CF algorithm for the Apache Spark platform was implemented in the Scala programming language.

II. The MapReduce paradigm

MapReduce is a paradigm for distributed programming, used for processing large amounts of data, with the use of groups of connected computer units, i.e. a computer cluster. Each computer unit is called *a node*. The paradigm was introduced by the Google company in 2004 [8]. Its main asset is that it takes away from the programmer most problems of parallel programming, such as node-to-node communication, management of cluster resources, and resistance to node failures. The MapReduce model is inspired by the *map* and *reduce* functions commonly used in functional programming.

A programmers task is to provide the implementation of the following two procedures (see Figure 1 for the MapReduce workflow):

- map() takes a fragment of the input data expressed as a set of pairs (key, value) and, on the basis of particular records, produces zero or more intermediate pairs. The MapReduce library groups all intermediate values related to the same intermediate key K. Next, the values are transferred to the Reduce() function.

$$Map(k1, w1) \longrightarrow list(k2, w2)$$

- reduce() takes the intermediate key K and a set of values for that key. Next, the values are joined. Each call to the Reduce function usually returns one value, but it can also return zero or more values.

$$Reduce(k2, list(w2)) \longrightarrow list(w3)$$

The MapReduce paradigm is based on the assumption that a large number of problems can be expressed as a finished set of tasks (called jobs) and each job consists of the map and reduce functions done in parallel.

The simplicity of the MapReduce paradigm makes it possible to effectively assign cluster resources to the performed jobs and to take away the complexity of their synchronization from the programmer but it also entails a lack of flexibility of programming: a program can only consist of subsequent map and reduce functions. Each phase of calculations can only be begun after the completion of each instance of the previous function. The only way to perform complex operations is to implement a few MapReduce jobs, the output data of which must be recorded on a hard disc and later uploaded again for the next job.

There have been several implementations of the MapReduce paradigm, but the Apache Hadoop technology has been the most successful for commercial purposes [9], [10].

III. Collaborative filtering

The collaborative filtering algorithm is used in modern recommendation systems and is very popular due to its universality (it is not dependent on the type of recommended items) and efficiency [11], [12], [13]. It utilizes the knowledge about the relationships between users and products for defining similarities in the evaluation methods, on the basis of which recommendations are made. With respect to the type of a considered similarity, the CF algorithm can be classified as user-based or item-based [14]. The systems implemented within the framework of this research used the item-based type of the algorithm in which products are recommended to a user (u) based on his or her previous purchases. The pseudocode of the algorithm is presented below.

```
for each i item for which u has not expressed a preference
    for each j item for which u has expressed a preference
        compute similarity s between i and j
        add the value of preference u for j with weight s
        to the calculated weighted average
return items with the highest weighted average value
```

To compute a similarity between two items, we propose the usage of the well-known Tanimoto coefficient (called also Jaccard) [15]. This coefficient was chosen because of the highest precision for possessed data. The Tanimoto similarity between two items i and j is defined as

$$s(i,j) = \frac{n(c_i \cap c_j)}{n(c_i \cup c_j)} = \frac{n(c_i \cap c_j)}{n(c_i) + n(c_j) - n(c_i \cap c_j)}$$

where $n(x)$ denotes the number of elements in the item-set (i.e. customer basket c).

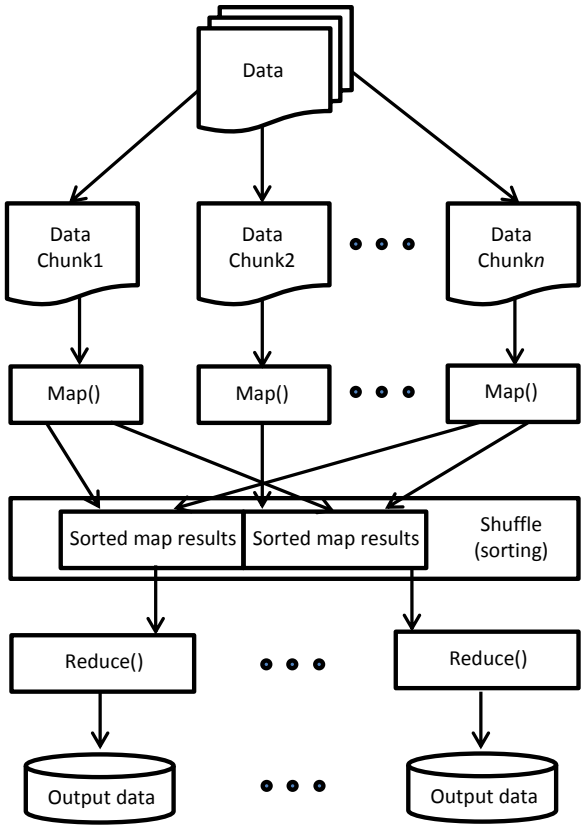

Fig. 1. The MapReduce model (based on [16])

IV. A paralleled version of collaborative filtering

In a paralleled version algorithm CF can be presented as a matrix equation:

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}_{ItemSim.} \times \begin{bmatrix} x \\ y \end{bmatrix}_{UserPrefs.} = \begin{bmatrix} xa & yb \\ xc & yd \end{bmatrix}_{UserRecs.}$$

in which recommendations (the recommendation vector) for a given user are calculated by multiplying the similarity matrix of items by the user vector (*Item Sim.* denotes Item Similarity, *User Prefs.* - User Preferences, and *User Recs.* - User Recommendations). The similarity matrix of items is a symmetrical matrix $n \times n$, where n is the number of items. Each element of the similarity matrix is a value of the similarity between the two objects.

A. The Hadoop solution

The Mahout library [6] provides implementations of paralleled versions of algorithms in the field of machine learning for the Hadoop platform. It focuses on classification, grouping, and collaborative filtering algorithms. The Mahout library was used for creating a recommendation system based on the Apache Hadoop technology.

In the Mahout library, the two most important programs which realize the paralleled CF algorithm based on items are *RecommenderJob* (which calculates recommendations) and *ItemSimilarityJob* (which calculates the similarity matrix).

A full implementation of the paralleled version of the collaborative filtering algorithm based on items, according to the MapReduce paradigm, is realized in the form of nine consecutive jobs. During the realization of the program, particular data from the disc are read nine times and then recorded, which has a significant impact on the reaction time of the system. That is why it was assumed that the implementation of the algorithm for the Spark platform would lead to greater efficiency.

B. Apache Spark

In April, 2012 (first works were conducted as early as 2010) the AMPLab at the University of California at Berkeley published an article [17] in which the idea of Resilient Distributed Datasets (abbreviated to RDD) was presented. The Spark platform is an open implementation of RDD. The aim of the creators of the Spark platform was to remove the limitations of the restrictive MapReduce paradigm. Spark can be viewed as a distributed computing platform in the same context as the open-source implementation of the MapReduce paradigm Hadoop [9]. Spark is fully compatible with HDFS [18].

RDD is a partitioned collection of data which means that particular elements of a collection can be divided among cluster nodes and processed in parallel. Additionally, we can distinguish the following characteristics of RDD:

- They can only be created by operating on other sets or while reading data from a file system because RDD sets cannot be modified.
- They are processed lazily and each set stores information about the operations made on input data, which makes it possible to recover a lost part of RDD in case of a node failure.
- Intermediate results of operations can be stored in cache memory so computations for iterative algorithms can be made much faster.

V. Experimental studies

The aim of the work was to implement and compare systems which generate recommendations with the use of distributed computing platforms, Hadoop, and Spark. Because of the limited time of access to the computing cluster the implemented systems were limited to computing the similarity matrices of items.

A. A description of the data

The data for the work were taken from an Internet portal in which registered users can use the available entertainment applications, such as games, quizzes, or tests. The portal users can like a given application. The preferences used for providing recommendations for users are expressed as Boolean values.

The set of data consists of 9,644,727 preferences (likes) expressed by 1,148,320 users with respect to 730 applications. The data are stored in the form of a text file. Particular preferences are separated by the new line sign (`\n`). A preference consists of a user identifier and an application identifier, separated by the tab sign (`\t`). User and application identifiers are unique integers. The size of the data is 106.5 MB.

B. A description of the experiment

All experiments were performed on a cluster which includes 3 nodes, one node as MasterNode and other 2 nodes as DataNodes. Each node was equipped with Intel Xeon L5640 processor (2.27 Ghz, 6 cores) and 6GB of RAM. The experiments run over Apache Hadoop Cloudera [19] version 4.1.3 (library Mahout version 0.7) and Spark version 0.7.3.

The authors original solution of paralleled item-based CF algorithm was implemented with the use of the Apache Spark platform and the Scala programming language [20].

The performance times depending on the amount of input data were measured in order to compare the efficiency of the implemented systems. Subsets differing with respect to the number of preferences were created for the experiment. The times taken to perform the computations of each system for particular data subsets were measured 10 times. The presented values are averaged.

Implemented solution consists of four major steps. The first step is to calculate the co-occurrences of items within the preferences of individual users.

```
val ratings = file.map(line => {
val fields = line.split("\t")
(fields(0).toInt, fields(1).toInt)
})
val ratings2 = ratings
val ratingPairs = ratings.join(ratings2).filter
{ case (user, (item1, item2)) =>
item1 < item2 }
```

Then, each item is assigned to the number of the expressed preferences. For optimization reasons, this set is distributed among all nodes via a variable of type Brodcast.

```
val numRatersPerItem = ratings.groupBy { case (user, item) =>
item }.map { case (item, ratingsPerItem) =>
(item, ratingsPerItem.size) }
val numRatersPerItemBC =
sc.broadcast(Map(numRatersPerItem.collect(): _*))
```

Each pair of co-occurring items is assigned to a list of proper users.

```
val usersForRatingPair = ratingPairs.map
{ case (user, (item1, item2)) =>
((item1, item2), user) }.groupByKey()
```

Using the received RDDs, it is possible to calculate the similarity between items.

```
val similaritiesInput = usersForRatingPair.map
{ case (pair, users) =>
(pair, (users.size, numRatersPerItemBC.value(key._1),
numRatersPerItemBC.value(key._2))) }
val similarities = similaritiesInput.map
{ case (pair, (size, numRaters1, numRaters2)) =>
(pair, jaccardSimilarity(size, numRaters1, numRaters2)) }
```

C. The results

Figure 2 shows the results of the research for a system which generates an item similarity matrix and is implemented in the Apache Spark technology, compared to the ItemSimilarityJob program from the Mahout library.

As the Spark platform has a very limited access to the cluster resources and there is no possibility of modifying the

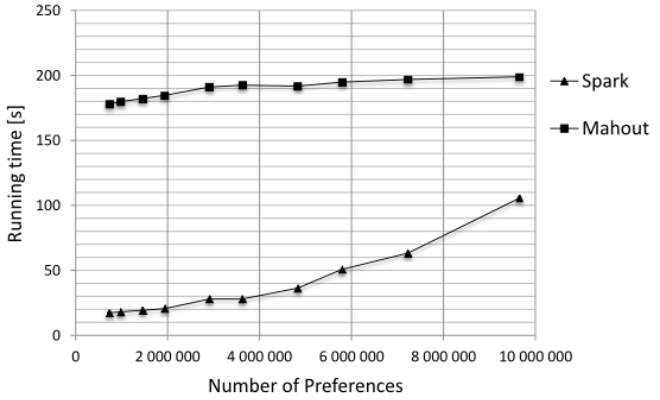

Fig. 2. A breakdown of the dependence of the performance time of computations on the amount of input data for ItemSimilarityJob and the Spark implementation

cluster configuration, the experiments have not been conducted with the full amount of possessed data. The most numerous dataset processed in its entirety was a set consisting of nearly 10 million of preferences. The averaged time of performing the computations in the ItemSimilarityJob program was 3 minutes 19 seconds, and in the Spark implementation – 1 minute 45 seconds, so the answer time was shortened almost by half in the case of the Spark platform. The lower the amount of input data, the greater was the difference.

D. Conclusions

It ought to be noted that, despite the considerable improvement of efficiency when the Spark platform was used, the results do not represent the whole picture. The direct reason for it is that the amount of input data for which the research was done was significantly limited the obtained characteristic of the Hadoop implementation does not show a linear dependence of time on the amount, and the increase of time between 700,000 and 10,000,000 preferences was only 20 seconds. Therefore, we can conclude that in the case of running a program consisting of five MapReduce tasks, which processed a small amount of data, most of the time was consumed by actions such as starting particular MapReduce jobs, synchronizing them, nodes communication, and reading and recording operations, and not directly for the computations. In order to obtain fully credible results, the measurements would have to be repeated on a greater scale.

VI. Conclusions and future work

Because of the limited time of access to the computer cluster used for the realization of the work, the item-based CF algorithm based on the Apache Spark technology was restricted to an item similarity matrix which can be used for generating recommendations on the fly, separately for each user. The Mahout library provides an analogous program named *ItemSimilarityJob* which was used for the comparative analysis.

Another problem encountered during the realization of the project was the limit of random access memory on the cluster for the Spark platform. Therefore, the measurements of the performance time of the computations dependent on the amount of input data were made for almost 10 million out of 145 million of the preferences present in the available data.

The conducted analyzes of the systems showed that the system based on the Spark platform was more efficient, as had been presumed. Although our implementation of Spark is still a prototype, early experience with the system is encouraging. We show that Spark can outperform Hadoop up to 10x for a smaller number of examined preferences (ca. 2 mln)! For full credibility, the research should be conducted with the use of a greater amount of data (and available cluster memory).

In the course of this work it was noticed that Apache Spark platform has features that justify its growing interest in the world of Big Data. In-memory based RDD paradigm allows for more efficient work allocation between nodes, reducing costly read/write operations of intermediate results on hard drives, which directly translates into increased jobs productivity. Furthermore, implementation of algorithms on Spark is much more developer-friendly, because there is no requirement to express jobs in successive mapping and reduction functions. Instead, the developer uses RDD transformations and actions, which are largely based on Scala's collections API. In our opinion, functional programming style seems to be more natural for parallel computing. Last but not least, Spark provides interactive shell (REPL), which allows for quick jobs prototyping, and encourages experimentation.

Further work on that topic should include completing the implementation of the collaborative filtering algorithm for the Spark platform and a repeating the analyzes with the use of a greater amount of data. Another possibility is to compare the obtained data with the results of implementations of other algorithms used in recommendation systems.

References

[1] C. Sammut and G. I. Webb, *Encyclopedia of Machine Learning*. Springer, 2011.

[2] (2014, November) IBM what is big data? Bringing big data to the enterprise. [Online]. Available: http://www.ibm.com/big-data/us/en/

[3] B. Sarwar, G. Karypis, J. Konstan, and J. Riedl, "Item-based collaborative filtering recommendation algorithms," in *Proceedings of the 10th international conference on World Wide Web*. ACM, 2001, pp. 285–295.

[4] Z.-D. Zhao and M.-S. Shang, "User-based collaborative-filtering recommendation algorithms on hadoop," in *Knowledge Discovery and Data Mining, 2010. WKDD'10. Third International Conference on*. IEEE, 2010, pp. 478–481.

[5] J. Jiang, J. Lu, G. Zhang, and G. Long, "Scaling-up item-based collaborative filtering recommendation algorithm based on hadoop," in *Services (SERVICES), 2011 IEEE World Congress on*. IEEE, 2011, pp. 490–497.

[6] S. Owen, R. Anil, T. Dunning, and E. Friedman, *Mahout in action*. Manning Publications Co., Greenwich, CT, USA, 2011.

[7] (2014, November) Apache mahout documentation. [Online]. Available: http://mahout.apache.org/

[8] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[9] (2014, November) Hadoop 1.1.2 documentation. [Online]. Available: http://hadoop.apache.org/docs/stable/

[10] T. White, *Hadoop: The definitive guide*. O'Reilly Media, Inc., 2012.

[11] P. Resnick and H. R. Varian, "Recommender systems," *Communications of the ACM*, vol. 40, no. 3, pp. 56–58, 1997.

[12] H. Tan and H. Ye, "A collaborative filtering recommendation algorithm based on item classification," in *Circuits, Communications and Systems, 2009. PACCS'09. Pacific-Asia Conference on*. IEEE, 2009, pp. 694–697.

[13] S. Gong, H. Ye, and H. Tan, "Combining memory-based and model-based collaborative filtering in recommender system," in *Circuits, Communications and Systems, 2009. PACCS'09. Pacific-Asia Conference on*. IEEE, 2009, pp. 690–693.

[14] R. V. Tatiya and A. S. Vaidya, "A survey of recommendation algorithms," *IOSR Journal of Computer Engineering*, vol. 16, no. 6, pp. 16–19, 2014.

[15] L. Kaufman and P. J. Rousseeuw, *Finding groups in data: an introduction to cluster analysis*. John Wiley & Sons, 2009, vol. 344.

[16] T. Kajdanowicz, W. Indyk, and P. Kazienko, "Mapreduce approach to relational influence propagation in complex networks," *Pattern Analysis and Applications*, pp. 1–8, 2012.

[17] 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 *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*. USENIX Association, 2012, pp. 2–2. [Online]. Available: http://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf

[18] (2014, November) Spark documentation. [Online]. Available: http://spark.apache.org/documentation.html

[19] (2014, November) Cloudera official web site. [Online]. Available: http://www.cloudera.com/content/cloudera/en/home.html

[20] (2014, November) Scala programming language. [Online]. Available: http://www.scala-lang.org