Development of a distributed recommender system using the Hadoop Framework

Raja Chiky, Renata Ghisloti, Zakia Kazi Aoul

LISITE-ISEP
28 rue Notre Dame Des Champs
75006 Paris
firstname.lastname@isep.fr

Abstract. Producing high quality recommendations has become a challenge in the recent years. Indeed, the growth in the quantity of data involved in the recommendation process pose some scalability and effectiveness problems. These issues have encouraged the research of new technologies. Instead of developing a new recommender system we improve an already existing method. A distributed framework was considered based on the known quality and simplicity of the MapReduce project. The Hadoop Open Source project played a fundamental role in this research. It undoubtedly encouraged and facilitated the construction of our application, supplying all tools needed. Our main goal in this research was to prove that building a distributed recommender system was not only possible, but simple and productive.

1 Introduction

The amount of information in the web has greatly increased in the past decade. This phenomenon has promoted the advance of the recommender systems research area. The aim of Recommender Systems is providing personalized recommendations. They help users by suggesting useful items to them, usually dealing with enormous amounts of data. Amazon, for example, that has incorporated recommender systems to personalize the online store for each user, has recorded in 2003 more than 29 million users and several million catalog items Linden et al. (2003).

Many recommender systems approaches have been developed in the past decade, but a considerable amount of them were constructed and evaluated with small datasets. Furthermore, the volume of web information has greatly increased in the last years, and for that, several recommender systems suffer from performance and scalability problems when dealing with larger datasets.

Our main goal in this paper is describing a method to possibly overcome these issues. We propose a distributed recommender system, and we intend to demonstrate that it could be easily developed and present good results. We chose The Slope One Lemire and Maclachlan (2005) as recommender algorithm and we study the MapReduce Dean and Ghemawat project to construct such distributed system.

MapReduce is a framework introduced by Google that supports distributed computing with large amount of data on a cluster of computers. An open source implementation of this framework is available in the Apache Hadoop had project.

In this paper, we describe the process of adapting the chosen recommender algorithm to Hadoop platform, and then, we verify its performance by comparing the distributed version with the standalone method.

The remainder of this paper is organized as follows. Section 2 describes the state of the art. In Section 3 , we present the global approach to construct a distributed recommender system. Section 4 presents the experimental study and the results. Finally, Section 5 concludes this paper and gives an outlook upon our ongoing and future research in this area.

2 Background:Slope One

Slope OneLemire and Maclachlan (2005) is a simple and efficient type of recommender algorithm. Introduced by Daniel Lemire and Anna Maclachlan in 2005, it involves a simpler idea than the majority of other collaborative filtering implementations. While these usually calculate the similarity between vectors of items using the cosine or the Pearson methods pea (1994), the Slope One approach recommends items to users based on the average difference in preferences of items.

The main idea of the algorithm is to create a linear relation between items preferences such as the relation

$$F(x) = x + b$$

The name "Slope One" comes from the fact that here the "x" is multiplied by "1". It basically calculates the difference between the ratings of items for each user (for every item the user has rated). Then, it creates an average difference (*diff*) for every pair of items. To make a prediction of the Item k for a User A for example, it would get the ratings that User A has given to other items and add to it the difference (diff) between each item. With this, we could obtain an average.

Being r_{Ai} the rating that user A has given to item i, *diff(i,j)*=r the difference between ratings of item i and item j, and supposing we have n items:

The prediction for the rating that user A could give to the Item k is given by

$$\frac{(r_{A1} + diff(k,1)) + (r_{A2} + diff(k,2) + ... + (r_{1n} + diff(k,n))}{n}$$

Below we present the pseudo-code version of the algorithm. It can be divided in two parts: the preprocessing and the prediction phase.

In the preprocessing phase, we calculate the difference between all item-item preference values

```
1. for every item i{
2.   for every other item j{
3.     for every user u rating both i and j{
4.             add the difference to
                an average diff(i,j)}}}
```

The prediction phase:

```
1.  for every item i not rated by a user u{
2.   for every item j rated by u{
3.    find diff(i,j)
4.     add this diff to u's rating for j}}
6.  return the top predicted items
```

In terms of execution, the most expensive phase is the preprocessing one, which can be precomputed. The algorithm is very attractive because its online portion, the prediction phase, is fast.

Its running time does not depend on the number of users, it depends mostly upon the average rating difference between every pair of items. Suppose we have m users and n items, computing the average differences could take up to mn^2 time steps. The storing of the $diff$ matrix could also be expensive. It could take up to $n(n-1)/2$ units of storage.

3 Approach

MapReduce is a framework introduced by Google for processing larges amounts of data. The framework uses a simple idea derived from the commonly known map() and reduce() functions used in functional programming (e.g. LISP). It divides the main problem into smaller sub-problems and distribute these to a cluster of computers. It then combines the answers to these sub-problems to obtain a final answer.

First, map() receives a set of key/value pair and produces an intermediary key/value pair as output. Then, these values are sorted by the framework and regrouped in a way that all values belonging to the same key are together (*combine()* function). Thus, as input to reduce() we have the same key supplied by map() with a list of corresponding values. This data is then processed by reduce() and a final output is produced.

The primary input data is chopped by the framework, which also is responsible for managing the intermediary data transition. For the user it remains only the task of defining the input/output interfaces and providing the map and reduce functions.

Based on this idea, we started by identifying an input format to create our own MapReduce Slope One Method. As a format we decide to use a typical type of a Text format, with a UserID (identification of the user), an ItemID (identification of the item) and a Rating from this user to this item.

The Slope One preprocessing phase can be divided in two parts. The first is when the items *diffs* are calculated for each user. In the second part, all *diffs* for every pair of items are added together to create an average *diff* between two items.

Our approach is as follows: First, the input data is divided in an independent way in order to regroup the ratings of the items using the users as keys. Then, the *diffs* for one user is computed. After that, all pair of intermediary *diffs* would be united to calculate the average *diff* for each pair of items.

Relying on this method two MapReduce separated processes are created: one that divides the input into independent chunks and calculates the item *diffs* for one user; and other one that having the lists of items *diff*, and calculates the overall *diff* between two items.

The first MapReduce presents a map() that receives the original input, produces a set of intermediary values containing each line of the input file - with a user, an item and a rating relating to these two. These values are sorted by the framework and then reduce() receives a user as key and a list with all his items and ratings. Reduce() calculates the *diffs* between every pair of items for this certain user and return them as output.

The second map() would be an identity function, meaning that the same value given as input would be given as output. On the other hand, the secondary reduce() would receive as key the item pair <itemi, itemj> and as values a list of all *diffs* related to this pair of items. It would be, then, a simple matter of calculating the average *diff* to every pair of items, and we would have as result a final list for every pair of items, containing the averages *diffs*.

4 Experimental evaluation

To our experimental tests the dataset used was the one provided by MovieLens mov (2003). MovieLens is a web-based recommender system where users rate movies and receive recommendations. Currently, it provides three packages of datasets: The fist one with 100,000 ratings for 1682 movies by 943 users, the second one with 1 million ratings for 3900 movies by 6040 users and the third one with 10 million ratings package for 10681 movies by 71567 users. These ratings represent the grades giving by MovieLens users to movies, varying from 1 (not liking a movie at all) to 5 (really liking the movie). The files contain information in the UserId, ItemId, Rating format. In this paper, we use the 1 million and the 10 million ratings packages.

For ensuring the correctness of the applications a forged test data was used; a small set of users and rating to enable a clear and simple comparison of the results. The dataset contains 5 users, 10 items and 28 ratings.

Both master and slave machines where the experiments took place had the same hardware specification and have the following characteristics: processor Intel 2.66 GHZ, 1.7GB of memory, and 100GB of disk. It runs on Ubuntu Linux 5.0. We use one master and two slaves in the case of fully distributed experimentation.

The first step in this experimental procedure was the execution of each SlopeOne approach with each dataset. Then, all output results were compared to check in the results matched. Finally, the execution time of the approaches were compared and analyzed.

The tests were made in two phases, one for each dataset. In each phase, we compare the execution time in milliseconds between every Slope One approach (standalone, pseudo-distributed and fully-distributed) and, of course, the uniformity of the final result.

The first experiment was with the small forged data. The output of all three approaches were compared to our hand-made and correct result. All three of them presented the correct output, proving the correctness of the algorithms. To generate a final execution time, each approach was executed three times in a row, and the result presented in the figure 1 (a) are the average of these three executions.

The second experimental phase was testing the 1 million MovieLens dataset for again all three approaches. The execution, as above, is an average of three single executions and the resuls is given in the figure 1 (b).

Finally, the last phase was testing the 10 million dataset from MovieLens. The Standalone operation was not able to deal with this amount of data, presenting a java exception which is java.lang.OutOfMemoryError. The other two approaches succeeded to finish the task and

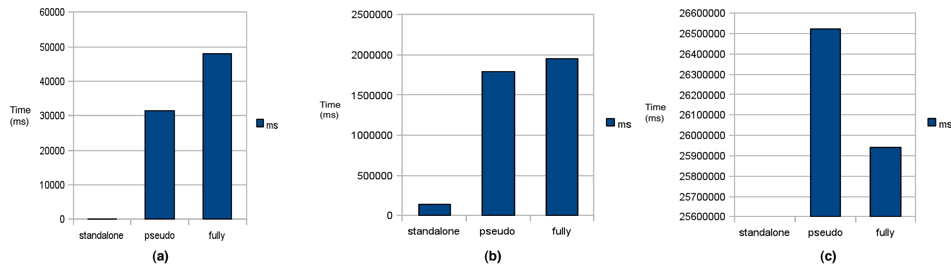

FIG. 1 – *Exectution time for the three datasets*

presented the average execution time represented in the figure 1 (c). From analyzing the results of the figure 1 it is possible to note that only from a certain point it becomes reasonable to use Hadoop. As every framework that provides abstraction layer, executing the framework has an initial cost. While testing the first and second dataset, it was visible that the ”weight ' of running Hadoop was greater than the benefits that it could bring. Hence, the running time of the standalone method was smaller than the other two approaches even thought these had more Java threads or computer-nodes working.

In the third scenario, we can see the fully-distributed approach overcomes the pseudo-distributed one. One possible reason for the pseudo doing better in the other scenarios is the cost involved in establishing a distributed cluster. With the increase in the amount of data, the advantages of a distributed cluster made difference, and provided a faster performance. Our main goal in this research was to verify if the distributed approach really brought any advantages to the recommender. Verifying the final results, it is clear that for a small dataset such as equal or inferior to 1 million ratings the standalone method is a better approach. However, it was proved that when dealing with larger amounts of data the Hadoop framework may be a feasible solution. When the volume of data grows, the fully distributed has a better performance than the other two methods. The standalone method wasn't able to perform at all in the third scenario and so couldn't give a result.

5 Conclusion

Recommender systems face a substantial challenge when dealing with huge amount of data. In this paper, our main goal was help solving this problem by describing an easy development of a distributed recommender system. This was possible by using powerful tools such as the Hadoop open source, MapReduce implementation and slopeone that is an efficient type of collaborative recommender algorithm.

The simplicity of Slope One added to the strong structure offered by Hadoop made possible to complete our task. Our results showed that indeed a distributed framework can provide good results, and, hopefully, encouraged the interest in this area of research.

Our future work will be to analyze other types of recommendation algorithms to study the possibility of making them effective on large data sets using Hadoop or other avalaibe frame-

works Nicolae et al. (2010). We also plan to consider a generic distribution of recommendation algorithms that requires no effort in rewriting the code.

References

Apache hadoop, http://hadoop.apache.org/.

(1994). *GroupLens: An Open Architecture for Collaborative Filtering of Netnews*. ACM.

(as of 2003). *MovieLens dataset, http://www.grouplens.org/data/.*

Dean, J. and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. pp. 137–150.

Lemire, D. and A. Maclachlan (2005). Slope One Predictors for Online Rating-Based Collaborative Filtering. In *Proceedings of SIAM Data Mining (SDM'05).*

Linden, G., B. Smith, and J. York (2003). Amazon.com recommendations: item-to-item collaborative filtering. Volume 7, pp. 76–80.

Nicolae, B., D. Moise, G. Antoniu, L. Bougé, and M. Dorier (2010). BlobSeer: Bringing High Throughput under Heavy Concurrency to Hadoop Map-Reduce Applications. In *24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010)*, Atlanta, États-Unis. IEEE and ACM.

Résumé

La quantité d'information dans leWeb a augmenté ces dix dernières années. Ce phénomène a favorisé la progression de la recherche dans le domaine des systèmes de recommandation. Ces systèmes ont l'intention d'aider les utilisateurs en fournissant des suggestions utiles. Nous proposons dans ce papier d'utiliser un algorithme de recommandation existant et de favoriser sa montée en charge.Nous utilisons pour cela le framework de développement Hadoop qui propose une implémentation du paradigme MapReduce pour la distribution des traitements. Notre principal objectif dans ce papier est de prouver que la construction d'un système de recommandation distribué est non seulement possible mais qu'elle est également simple et bénéfique.