Cosine similarity for collaborative filtering: a distributed algorithm with Spark in the cloud

Giulia Corsi
DISI-University of Trento
Via Sommarive, 9
38123 Trento, Italy
giulia.corsi@studenti.unitn.it

Gianvito Taneburgo
DISI-University of Trento
Via Sommarive, 9
38123 Trento, Italy
gianvito.taneburgo@studenti.unitn.it

ABSTRACT

In this paper we propose a distributed algorithm to efficiently compute cosine similarities between vectors in a high-dimensional space defined by a big data set. The solution can be useful for recommender systems using collaborative filtering techniques, for example those computing user-to-user similarities between customers of big on-line shops. The algorithm has been designed following the MapReduce model in order to make it compatible with any framework supporting this functional paradigm. An implementation has been realized for Spark, a large-scale data processing engine. By using Spark's API for Python, PySpark, we have been able to write a concise and expressive implementation of the algorithm in less than one hundred lines of code, ready to be executed by the framework on a cluster of machines. Thanks to this tool, it has been possible to compute in a reasonable time cosine similarities between users contained in the Netflix Prize training data set [3], that provides more than 100 millions ratings for movies and tv series. A single computer would not have been able to complete the same task in a comparable time. The performances of the algorithm have been tested on different on-line clusters of machines rented using Cloud Dataproc, a Spark SaaS (Software as a Service) offered by Google Cloud Platform.

CCS Concepts

•**Computing methodologies** → **MapReduce algorithms;**
•**Computer systems organization** → *Cloud computing;*
•**Information systems** → *Retrieval efficiency;* Recommender systems;

Keywords

Cosine similarity; distributed algorithm; Spark; cloud; Google Cloud Platform; Cloud Dataproc; Compute Engine

1. INTRODUCTION

With the success of social media and on-line shopping companies such as Facebook, Amazon and Netflix, the new millennium has been characterized by a massive growth of the quantity of data to analyze. This has been the origin of new computer science branches and an incentive to launch distributed platforms dedicated to big data storage and analysis, for which old centralized systems are not sufficient any more. A distributed system is characterized by many components (cluster of commodity machines) that communicate together in order to coordinate their actions and solve hard computational problems [4]. Distributed systems present many challenges, among which there are scalability, failure handling, heterogeneity, security, load balancing, concurrency and performance.

One of the purposes of data analysis regards recommender systems, which estimate ratings for items that have not been seen by a user, according to ratings given by other users and some other information [2]. This analysis is related to big data when applied on huge data sets. An excellent recommender system has a key role for on-line companies since it responsible of suggesting items (products for Amazon, movies for Netflix, ads for Google and Facebook, etc.) that could lead to huge earnings or loss of money or users.

Contribution. In this paper we focus on how to exploit some tools and platforms for distributed computation in order to calculate the cosine similarity between vectors of ratings representing users, derived from the big Netflix training data set. The size of the data set [3], containing more than 100 millions of ratings, makes unfeasible processing this task on a single machine. The similarities outputted by the algorithm could be later used by recommender systems in the form of an external precomputed knowledge instilled into the application to enable, for an example, an hybrid recommendation approach exploiting some content-based algorithm. Even if the algorithm has been tested on this specific domain of user ratings for movies, its main idea can be easily reused in different contexts without too much effort. The computation of cosine similarities between users is only one of its many possible use cases.

Related work. With the continuous growth of data available for analyses, a lot of efforts have been done to find distributed solutions for data storage and processing. In the last five years Hadoop has been one of the most popular platforms to build big data solutions in many different fields. This project, written in Java, has been developed by the Apache Software Foundation and it offers two scalable services: a distributed file system (Hadoop Distributed File System, HDFS), designed to store huge quantity of data, and an implementation of the MapReduce framework to efficiently process data coming from different sources, even in a streaming fashion. Hadoop can be used on a single machine, with pseudo-distribution of the workload (useful to test and debug applications), or on a cluster of commodity hardware to take advantage of the potential of parallel computation.

Many other tools have been built on the top of Hadoop to simplify routinely tasks or to allow some other types of operations on data residing on the distributed file system, such as

SQL-like queries implemented with the MapReduce framework. In 2014 AMPLab, at University of California, has developed Spark, another open source framework for distributed computation, and then donated it to the Apache Software Foundation. Thanks to its in-memory primitives, Spark is able to complete workloads on cluster of machines faster than Hadoop. On some kind of tasks it is one hundred times faster than Hadoop [7]. We have decided to adopt this framework to implement our algorithm not only because of its speed in large-scale data processing tasks, but also because Spark supports multiple programming language (Java, Scala, Python, R), it is well documented and easy to use, since it takes over the management of cluster routines and configuration.

Structure of the paper. The paper is structured as follows. In Section 2 the MapReduce model is briefly explained and the main differences between Hadoop and Spark are highlighted to better comprehend the algorithm. Section 3 gives some details about the input and output data sets. Section 4 describes the cosine similarity measure and how the input data set has been represented in terms of vectors. Section 5 is dedicated to the problem formulation and solution, the distributed algorithm on Spark. Section 6 provides some insights about the Google Cloud Platform technologies that have been used to test the algorithm on clusters of machine. Section 7 discusses the experiments and the results achieved. Conclusions on our work are in Section 8 of the paper.

2. HADOOP AND SPARK COMPARISON

Hadoop owes a great portion of its notoriety to Google's MapReduce programming model, whose principle has guided the design of the framework core. In the same way, Spark relies on Hadoop, making it an abstract layer providing some cluster management services, like those related to workloads balancing, nodes status monitoring, and bookkeeping. Even if the two tools share many features and are inspired by the same concepts, they are very different from some points of view. These differences have been determining both for the choice of the tool to use for this work and for the design of our algorithm, so it is worth explaining them.

Hadoop is only capable to execute in a distributed manner programs closely fitting the MapReduce paradigm, those made up of user-defined map and reduce functions. During the first phase, the map function is applied by every node of the cluster, named *worker*, to the block of input data he owns, named *chunk* (note that data is split into the distributed file systems across all the cluster nodes). The map function takes as input a key-value pair (K, V) and outputs a list of intermediate key-values pairs (I, P). Then, the framework automatically groups the intermediate pairs sharing the same key I and collapse their values P into a list. Once this process is over, the reduce function is executed. It takes as input one of these pairs $(I, [P1, P2, P3, ...])$ provided by the framework and outputs a possibly smaller set of values, for example by applying an aggregating operation on $P1, P2, P3, ...$

$$map(K, V) - - > list(I, P)$$
$$reduce(I, list(P)) - - > list(P)$$

The framework automatically assigns to the available workers the execution (and, in case of failure, the re-execution) of map and reduce functions and collects their results as soon as they are available (user can customize and optimize this step by overriding Hadoop default behavior). Each node, in fact, has a copy of the source code of both operators that can be executed upon requested. A *master node* is responsible of orchestrating the execution of such workflow.

User programs can also consist of a sequence of map-reduce pairs that are executed one-by-one, sequentially. One of the limits Hadoop is often blamed for is the strict requirement of both and only these functions. There is no way for a user to avoid specifying either the map or the reduce function. The best that can be done is writing an identity function to preserve the output of the other operator (still there is a waste of resources since the identity function will be uselessly executed multiple times causing overhead). In the same way, no other operators is allowed. Every operation must be emulated by a map-reduce cycle. Although this is almost always possible, for sure this is not a flexible mechanism, as stated by Google when MapReduce has been described for the first time to general public. [5]

Spark overpasses these limits and provides a rich and powerful API featuring many different operators. By combining these functions it is possible to easily write complex applications. The typical result is a pipeline of different operators, including map and reduce functions in any order and multiplicity. Some of these operators can be executed on the partition of data being held by the worker, others require a shuffling of chunks between the nodes of the cluster, thus resulting in slower execution times. Knowing which operators are faster than others is essential to achieve good performances. A typical Spark application is usually composed of two parts. In the first step input data are gathered into a Resilient Distributed Dataset (RDD), the abstraction Spark uses to spread read-only data between cluster nodes. In the second step a series of transformations is applied to the RDD via a user-defined pipeline of operators. Being the RDD read-only, each of these operators produces a new modified RDD. Spark's scheduling engine is able to optimize this process by combining some operators. In fact, the pipeline is lazily evaluated and operators are applied on the RDD only when a result is requested by the user.

Our distributed algorithm takes advantage of this great flexibility and, for such reason, uses a pipeline of operators different from a map-reduce cycle. This also means that some tweaks are required in order to make it runnable on a traditional Hadoop cluster rather than on a Spark one.

3. INPUT AND OUTPUT DATA SETS

The input data set used for the algorithm is derived from the training set provided by Netflix during the Netflix Prize competition held in 2006. It is made up of 100,480,507 ratings that 480,189 anonymous users gave to 17,770 different movies or tv series up to 2005 [3]. The original directory containing these data was composed of one file per movie. Every file contained a line with a unique identifier for the movie and a set of subsequent lines, each one corresponding to a rating from a customer. The format was the following:

<CustomerID,Rating,Date>

where CustomerIDs ranged from 1 to 2,649,429 (with gaps since users are 480,189) and ratings were on a five star scale from 1 to 5 (integral). Not being interested in information related to the date in which the rating occurred, the data

set has been processed in order to only keep lines in the following format:

<CustomerID,Rating>.

The desired output of the algorithm is a directory containing files in the following format:

CustomerID1#CustomerID2
CosSim(CustomerID1, CustomerID2)
CustomerID1#CustomerID3
CosSim(CustomerID1, CustomerID3)
CustomerID4#CustomerID6
CosSim(CustomerID4, CustomerID6)
CustomerID3#CustomerID7
CosSim(CustomerID3, CustomerID7)
...

where:

- CustomerIDi and CustomerIDj are users having co-rated at least two movies, with CustomerIDi having a lower identifier than CustomerIDj.
  '#' is a separator character useful during parsing of the output data set;

- CosSim(CustomerIDi, CustomerIDj) is the similarity value calculated between the two users according to co-rated movies.

4. COSINE SIMILARITY

Before presenting the distributed algorithm, the standard formula to compute the cosine similarity between two vectors is reported.
Given two vectors of attributes, $\vec{u}$ and $\vec{v}$, the cosine similarity between the two vectors, $\cos(\theta)$, is represented using a dot product and magnitude as:

$$\cos(\theta) = \frac{\vec{u} \cdot \vec{v}}{\|\vec{u}\|\|\vec{v}\|} = \frac{\sum_{i=1}^{n} u_i v_i}{\sqrt{\sum_{i=1}^{n} u_i^2}\sqrt{\sum_{i=1}^{n} v_i^2}}$$ [6],

where u_i and v_i are components of vectors $\vec{u}$ and $\vec{v}$ respectively representing their attributes.
Netflix users can be modeled as vectors of a high-dimensional space. Such space has size equals to the number of different movies and tv series in the data set. Each user is represented as a vector having, for each component, the rating he has given to the corresponding movie. Since every user has rated only few items, these vectors are very sparse. Netflix ratings range from 1 to 5 and 17,770 items are present in the data set, thus for each customer we have a vector $\vec{u}$ such that

$$\vec{u} \in \{0, 1, 2, 3, 4, 5\}^{17780}.$$

Given two vectors representing users $\vec{u}$ and $\vec{v}$, their similarity is given by the following formula:

$$\text{sim}(\vec{u}, \vec{v}) = \frac{\sum_{m \in M_{\text{u,v}}} r_{u,m} r_{v,m}}{\sqrt{\sum_{m \in M_{\text{u,v}}} r_{u,m}^2}\sqrt{\sum_{m \in M_{\text{u,v}}} r_{v,m}^2}},$$

where $M_{\text{u,v}}$ is the set of items co-rated by the two users u and v, and $r_{x,y}$ is the rating given by user x to item y.
It is clear that this similarity measure is symmetric, since the angle between the two vectors is the same regardless the order in which such vectors are considered, that is

$$\text{sim}(\vec{u}, \vec{v}) = \text{sim}(\vec{v}, \vec{u}) = \cos(\theta).$$

To improve the performances of the algorithm, the output data set contains only one entry for each pair of customers. If the cosine similarity has been computed for the pair (CustomerIDx, CustomerIDy), than there is no output entry for the pair (CustomerIDy, CustomerIDx).

5. DISTRIBUTED ALGORITHM FOR SPARK

During our work, many different algorithms have been tested to compute the cosine similarities between vectors in a distributed manner with Spark. This task, in fact, can be accomplished in multiple ways, but experiments have proven some solutions to be much more efficient than others. Obviously, only the fastest algorithm is presented in this section. Such algorithm takes advantage of the associative property of addition and multiplication (recall numerator and denominator of the cosine similarity formula previously reported) to avoid using Spark operators requiring heavy shuffling of data chunks between workers.
Like the vast majority of Spark applications, our solution is composed of two parts: the RDD building and the processing pipeline. In order to simplify the comprehension of the algorithm, an example is provided and the RDD transformations are highlighted step by step. From now on, it is supposed that our input data set is composed of ratings provided by the following text files:

102.txt	**164.txt**	**1162.txt**
55555,2	22222,3	66666,3
22222,4	11111,3	22222,3
11111,3	55555,4	11111,3
33333,2	44444,5	44444,4
44444,1		

The RDD is built first by loading in the master's memory (part of) the data set and then by splitting the input into partitions that are spread between workers. Each node now contains some key-value pairs, with keys being strings representing file names and values being strings containing ratings:

[On partition 1]
('102.txt','55555,2\n22222,4\n11111,3\n33333,2\n44444,1')

[On partition 2]
('164.txt', '22222,3\n11111,3\n55555,4\n44444,5')
('1162.txt', '66666,3\n22222,3\n11111,4\n44444,4')

The pipeline can be now executed. It is composed of 4 steps:

1. *Quadruple generation* [operator: *flatMap*].
   Ratings are parsed from the values of each pair in the data set. Then, for each possible couple of users, a new key-value pair is added to the RDD. The key is a combination of two users identifiers (smallest identifier, # separator, biggest identifier) and the value is a quadruple of integers. The first integer represents how many movies are co-rated by the two users, so it is always 1 in this step; the second is the product of the two

ratings given by the users to the movie (addend in the numerator of the cosine similarity formula); the third and the fourth are the squares of such ratings (factors in the denominator of the cosine similarity formula). The total number of pairs produced for a file with n rating is

$$\frac{n(n+1)}{2}.$$

$\mathbf{f}\ (string, string) \longmapsto list(string, int, int, int, int)$

$\mathbf{f}\ (file\ name, file\ content) \longmapsto$
$[id_j \# id_k, (1, r_j r_k, r_j^2, r_k^2)]$

[On partition 1]

[For the pair with key '102.txt']
('11111#22222', (1, 12, 9, 16))
('11111#33333', (1, 6, 9, 4))
('11111#44444', (1, 3, 9, 1))
('11111#55555', (1, 6, 9, 4))
('22222#33333', (1, 8, 16, 4))
('22222#44444', (1, 4, 16, 1))
('22222#55555', (1, 8, 16, 4))
('33333#44444', (1, 2, 4, 1))
('33333#55555', (1, 4, 4, 4))
('44444#55555', (1, 2, 1, 4))

[On partition 2]

[For the pair with key '164.txt']
('11111#22222', (1, 9, 9, 9))
('11111#44444', (1, 15, 9, 25))
('11111#55555', (1, 12, 9, 16))
('22222#44444', (1, 15, 9, 25))
('22222#55555', (1, 12, 9, 16))
('44444#55555', (1, 20, 25, 16))

[For the pair with key '1162.txt']
('11111#22222', (1, 9, 9, 9))
('11111#44444', (1, 12, 9, 16))
('11111#66666', (1, 9, 9, 9))
('22222#44444', (1, 12, 9, 16))
('22222#66666', (1, 9, 9, 9))
('44444#66666', (1, 12, 16, 9))

2. *Quadruple aggregation* [operator: *reduceByKey*]. The quadruples outputted in the previous step belonging to the same key (pair of users) are aggregated, first in the same partition, then across partitions. Each element in the first quadruple is added to the corresponding one in the second quadruple. In this way, the first element keeps working as counter of co-rated movies, while the other three become, in the end, the three summations present in the cosine similarity formula.

$\mathbf{f}\ (int, int, int, int, int, int, int, int) \longmapsto$
(int, int, int, int)

$\mathbf{f}\ ((count1, sum1, fac1, fac2),$
$(count2, sum2, fac3, fac4)) \longmapsto (count1 +$
$count2, sum1 + sum2, fac1 + fac3, fac2 + fac4)$

[On partition 1]

[No reduction can be executed]

[On partition 2]

[For pairs with key '11111#22222']
(1, 9, 9, 9)
(1, 9, 9, 9)
⇓
(2, 18, 18, 18)

[For pairs with key '11111#44444']
(1, 15, 9, 25)
(1, 12, 9, 16)
⇓
(2, 27, 18, 41)

[For pairs with key '22222#44444']
(1, 15, 9, 25)
(1, 12, 9, 16)
⇓
(2, 27, 18, 41)

[No reduction involves the other pairs]
('11111#55555', (1, 12, 9, 16))
('11111#66666', (1, 9, 9, 9))
('22222#55555', (1, 12, 9, 16))
('22222#66666', (1, 9, 9, 9))
('44444#55555', (1, 20, 25, 16))
('44444#66666', (1, 12, 16, 9))

[Pairs from the two partitions get shuffled into one...]

[On some partition]

[For pairs with key '11111#22222']
(2, 18, 18, 18)
(1, 12, 9, 16)
⇓
(3, 30, 27, 34)
[For pairs with key '11111#44444']
(2, 27, 18, 41)
(1, 3, 9, 1)
⇓
(3, 30, 27, 42)
[For pairs with key '11111#55555']
(1, 12, 9, 16)
(1, 6, 9, 4)
⇓
(2, 18, 18, 20)
[For pairs with key '22222#44444']
(2, 27, 18, 41)
(1, 4, 16, 1)
⇓
(3, 31, 34, 42)
[For pairs with key '22222#55555']
(1, 8, 16, 4)
(1, 12, 9, 16)
⇓
(2, 20, 25, 20)
[For pairs with key '44444#55555']
(1, 20, 25, 16)
(1, 2, 1, 4)
⇓
(2, 22, 26, 20)

3. *Similarity computation* [operator: *flatMap*].
   The cosine similarity for each pair of users is calculated by using the last three values in the corresponding quadruple. If the counter (first element in the quadruple) is equals to 1 (no reduction has been applied to the key in the previous step), then the entire key-value pair is dropped. The reason for such choice is that the computed value would be meaningless: the cosine similarity between vectors in a one-dimensional space is always 1.

$$\mathbf{f}\ (string, int, int, int, int) \longmapsto (string, double)$$

$$\mathbf{f}\ (id_j\#id_k, (count, sum, fac1, fac2)) \longmapsto (id_j\#id_k, sum/(\sqrt{fac1}\sqrt{fac2}))$$

[On some partition]

[For the pair with key '11111#22222']
(3, 30, 27, 34) –> $30/(\sqrt{27}\sqrt{34}) = 0.990147542977$
[For the pair with key '11111#44444']
(3, 30, 27, 42) –> $30/(\sqrt{27}\sqrt{42}) = 0.890870806375$
[For the pair with key '11111#55555']
(2, 18, 18, 20) –> $18/(\sqrt{18}\sqrt{20}) = 0.948683298051$
[For the pair with key '22222#44444']
(3, 31, 34, 42) –> $31/(\sqrt{34}\sqrt{42}) = 0.820346992239$
[For the pair with key '22222#55555']
(2, 20, 25, 20) –> $20/(\sqrt{25}\sqrt{20}) = 0.894427191$
[For the pair with key '44444#55555']
(2, 22, 26, 20) –> $22/(\sqrt{26}\sqrt{20}) = 0.964763821238$

[Discarded pairs]
('11111#66666', (1, 9, 9, 9))
('11111#33333', (1, 6, 9, 4))
('22222#33333', (1, 8, 16, 4))
('22222#66666', (1, 9, 9, 9))
('33333#44444', (1, 2, 4, 1))
('33333#55555', (1, 4, 4, 4))
('44444#66666', (1, 12, 16, 9))

4. *Output storing* [operator: *saveAsTextFile*].
   Values are stored into multiple files with a utility function provided by Spark. The result is described in Section 3, the one regarding the input and output data sets.

6. EXECUTION IN THE CLOUD

In order to test the performances of the algorithm, clusters different for type and number of nodes have been required. Not being interested in the typical and sometimes painful initial clusters tuning process and lacking a reasonable number of machines, we have decided to try some out-of-the-box products, leveraging on some ad-hoc enterprise solutions. In particular, for our experiments we have decided to use some components of the Google Cloud Platform: Cloud Storage, Compute Engine and Dataproc, the latter still in a beta version. A brief overview of these products is presented.

6.1 Cloud Storage

Cloud Storage is an on-line repository of files, that get grouped in buckets having unique worldwide URIs. Files can be organized in folders and sub-folders, like on a traditional file system, similarly to many other file hosting services (Dropbox, Google Drive, etc.). Different permission policies allow files to be downloaded by specific groups of users or to be globally available to anyone in possession of their links. The Python script containing the source code of the algorithm implementation has been uploaded to Cloud Storage and made available to every cluster node. At the beginning of our experiments Cloud Storage has been designed to also host the input and output data sets. Spark master node, in fact, was supposed to download the input data, process it by storing intermediate and final results into the HDFS and, finally, upload the cosine similarities files to Cloud Storage. This choice has been proved to be inefficient since creating two bottlenecks, the first while downloading the set of files and the second while uploading the computed ones. Since many tests were going to be executed, we have decided to opt for a solution much more efficient on the long run, even if slightly more cumbersome, by using Compute Engine permanent disks.

6.2 Compute Engine

Compute Engine is the Google's service designed to provide infrastructures, offering high performance virtual machines and miscellaneous resources, with fast networking and a customer-friendly pricing. Spark clusters deployed via Cloud Dataproc rely on virtual machines spun up and transparently managed by Compute Engine. As just said, this service has been only directly used for permanent disks. Compute Engine, in fact, enables developer to create disks of different size and speed that can be freely attached and detached to virtual machines. In our case, a permanent disk containing the input data set has been attached to every master node before the execution of the algorithm and detached after having copied the output files. In this way there has been no need to download over and over data before every experiment.

6.3 Cloud Dataproc

The last and most important service of the Google Cloud Platform that has been used is Cloud Dataproc. This service allows developers to rent pre-configured Spark clusters differing for machine type and number. Configuration scripts can be used to execute ad-hoc commands on each machine when spun up. Anyway, nodes can be easily managed after the deploy by connecting to them via SSH. The cluster is capable of executing Spark tasks without any additional effort. Jobs can be submitted to the cluster via command line or via web interface. Their status can be checked at any time and statistics and logs get collected automatically.

7. EXPERIMENTS AND RESULTS

The algorithm previously presented has been tested on different cluster configurations to check how its performances vary in relation to the data set input size and the cluster computational power. In particular, during the first round of experiments the same chunk of data has been processed by clusters of increasing computational power. In the second phase, on the contrary, the same cluster has processed data sets of increasing size.
Before reporting results it is worth providing some details on the virtual machine types that have been used during tests. Compute Engine offers some general purpose machines, identified with the keyword *standard*, some others

Table 1: Compute Engine instance types overview

Instance identifier	Virtual CPUs[1]	Memory (GB)
n1-standard-4	4	15
n1-standard-8	8	30
n1-standard-16	16	60
n1-highmem-4	4	26
n1-highmem-8	8	52
n1-highmem-16	16	104

specific for memory intensive tasks, labeled with *highmem*, and, finally, some for heavy CPU workloads, the *highcpu* ones. [1] Since the most critical part of the algorithm is the generation of millions vectors pairs that could have introduced memory bottlenecks, only *standard* and *highmem* instances have been used. Table 1 offers an overview of the instance types of interest.

During some preliminary tests we have tried to process the entire data set on different high-profile cluster configurations. None of these has succeeded in completing the task. The reason is that some Netflix movies have been rated by over 15,000 users thus causing the algorithm to produce more than 200 millions of pairs during its first step for each of these movies. It has been immediately clear that with the available resources we would have never been able to process all the data set in a single step. Further considerations on this issue are later presented in conclusions. For such reason we have decided to sort files by increasing size and to process only the smallest one, those generating the lowest number of pairs. It is worth noting that by considering many small files we have not oversimplified the task at all: the number of pairs produced by each file has been decreased, but the total number of pairs has been nevertheless big when a sufficiently large sample has been considered. From now on, when referring to a part of the input data set of a specific size we will suppose such data set to be composed of the smallest files whose total size is greater or equal than the threshold.

The experiments presented in this paper were supposed to be replicated five times, in order to show, in this section of the paper, both the average execution time and the best one. While performing our tests, however, we have encountered some issues. We have noticed, in fact, that performances started to get worse after the first execution. Running times were usually 5-6 times slower from the second attempt and the number of communication errors between nodes increased, as confirmed by the huge amount of warnings logged by the master. We suppose Spark clusters to have been improperly reset by Cloud Dataproc after each execution. Some processes must have survived after each execution, by probably causing some memory leaks in RAM or swap memory. This malfunctioning could be justifiable if we consider the beta status of Cloud Dataproc. For such reason all the results presented refer to the first execution, the only one unbiased by the environment.

As a final note we also report that every experiment has been performed without setting any cluster parameter. All the tests, in fact, have been executed with the default settings provided by Google that, although probably well-tuned, may not be the optimal ones. This means that, although execution times are very encouraging, better results could be achieved.

[1]A virtual CPU is implemented as a single hardware hyperthread on a 2.6GHz Intel Xeon E5 (Sandy Bridge), 2.5GHz Intel Xeon E5 v2 (Ivy Bridge), or 2.3 GHz Intel Xeon E5 v3 (Haswell)

Table 2: Experiment 1: 10 MB data set

Master type	Workers type	Cluster memory	Running time (s)
n1-standard-4	3x n1-standard-4	90 GB	709
n1-standard-8	5x n1-standard-4	105 GB	489
n1-standard-16	7x n1-standard-4	165 GB	283
n1-highmem-8	6x n1-highmem-8	364 GB	211

7.1 First experiment: cluster performances

In the first round of experiments, as previously said, the same chunk of data has been processed on different clusters. A chunk of 10 MB (6097 movies) has been chosen as sample, supposing that it would have been processed in reasonable times even by low-profiles clusters. Four different configurations of nodes have been tested, by varying both the instance types (*standard* and *highmem*) and the number of workers (from 3 to 7). We have tried to cover a wide range of setups with the lowest number of master/workers combinations.

Results, reported in Table 2, have confirmed our prevision. Every cluster has succeeded in processing the chunk of data, even if the total amount of vectors (users) pairs to be processed was more than 82 millions. Execution times have reasonably proven to be inversely proportional to the total amount of RAM available between cluster nodes. The ranking does not change even if we consider, instead of the absolute amount of cluster memory, the ratio between RAM available and execution time (GB/s).

The last cluster configuration used has been the best one we could afford with our Google Cloud Platform account. It would have been interesting to see up to which cluster configuration such proportionality law would have held. We believe that at a certain point performances would not have improved any more and we suppose this point to correspond to the moment in which each node has enough memory to store every possible users pairs, thus avoiding swapping data with disks.

7.2 Second experiment: cluster scalability

For the second experiments we have adopted the inverse methodology, by fixing the cluster configuration and by processing an increasing amount of data. In order to be able to process the largest data set as possible the best available cluster has been used, the one having a n1-highmem-8 node as master and 6 similar nodes as workers. Before starting the experiment, we expected to see processing times increasing proportionally to the input size. Experiments have con-

Table 3: Experiment 2: 7x n1-highmem-8 cluster

Data set size (MB)	Movies number	Pairs number	Running time (s)
5	3918	30,403,437	34
10	6097	82,678,761	211
15	7540	160,970,128	404
20	8593	267,921,216	859
40	11019	1,043,445,947	3859

firmed our suppositions and results are presented in Table 3, together with the numbers of processed pairs. It is worth noting that, during the last experiment, more than one billion pairwise similarities have been computed in circa an hour.

8. CONCLUSIONS

From the experiments presented so far we can draw some conclusions on different topics, including Google Cloud Dataproc, Spark and the distributed algorithm presented.

We will consider, first of all, Cloud Dataproc. The product has proven to be stable enough, despite its beta status. Such behavior is consistent with Google's attitude to flag as beta every product almost production-ready. The clean-up issues reported in the previous section represent the only drawback we have experienced during our tests. Although being annoying, shutting down and deploying new clusters has not compromised our tests, at most our wallets. On the other side, Cloud Dataproc takes advantage of the entire ecosystem of products bundled with the Google Cloud Platform and performing tedious tasks such as deploying clusters or creating, attaching and (un)mounting disks to virtual machines can be easily done with few commands. Finally, the convenience of not having to manually configure any node of the cluster is, in our opinion, sufficient to foster the technology.

In spite of being a relatively new product, Spark has been able to complete our tasks in reasonable times. By inspecting execution logs its optimization engine has proven to be able to minimize data shuffling between reduce operations, causing an high overall throughput. Logging information from workers has been a little by cumbersome, just like discovering which operators require more time and resources to be executed. We think, however, these problems to be caused by our inexperience, since it was the first time we used this framework. Concluding, we also believe that the communication errors described before are not ascribable to Spark but to its underlying layers, Hadoop and Cloud Dataproc sandbox. Our final judgment on the framework is overall positive.

Finally, we present some conclusions on our distributed algorithm to compute cosine similarity. Before starting our work we were conscious that the problem was computationally hard due to the size of data to be processed, but we did not honestly imagine it would have been so tough. Computing the cosine similarity between two vectors is indeed a simple task, that do not require heavy mathematical operations, but doing such operation for all the users in the data set was impossible with our resources. In fact, producing all the possible users pairs for each file would have implied generating a massive amount of data (2,824,541,593,769 vectors pairs, to be precise) impossible to keep in memory. Our approach works well if the dataset, despite its total size, is made up of relatively small files, but can not be applied without additional improvements to other scenarios. This is the reason why this approach is generally avoided by big online companies in favor of other types of algorithms, like the one involving big matrices decomposition into smaller one, preserving some desired properties. It is not a coincidence if, for these types of tasks, Netflix uses clusters of hundreds machines and takes advantage of the higher computational power of GPUs instead of CPUs.

9. REFERENCES

[1] Google compute engine - instance types. *https://cloud.google.com/compute/docs/machine-types*.

[2] G. Adomavicius and A. Tuzhilin. Toward the next generation of recommender systems: A survey of the state-of-the-art and possible extensions. *IEEE Transactions on knowledge and data engineering*, 2005.

[3] J. Bennett and S. Lanning. The netflix prize. *KDD Cup and Workshop*, August 2007.

[4] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair. *Distributed systems: Concepts and Design*. 2012.

[5] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *OSDI - Operating Systems Design and Implementation*, 2004.

[6] J. Leskovec, A. Rajaraman, and J. Ullman. *Mining of Massive Datasets*.

[7] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: Sql and rich analytics at scale. *SIGMOD International Conference of Management*, June 2013.