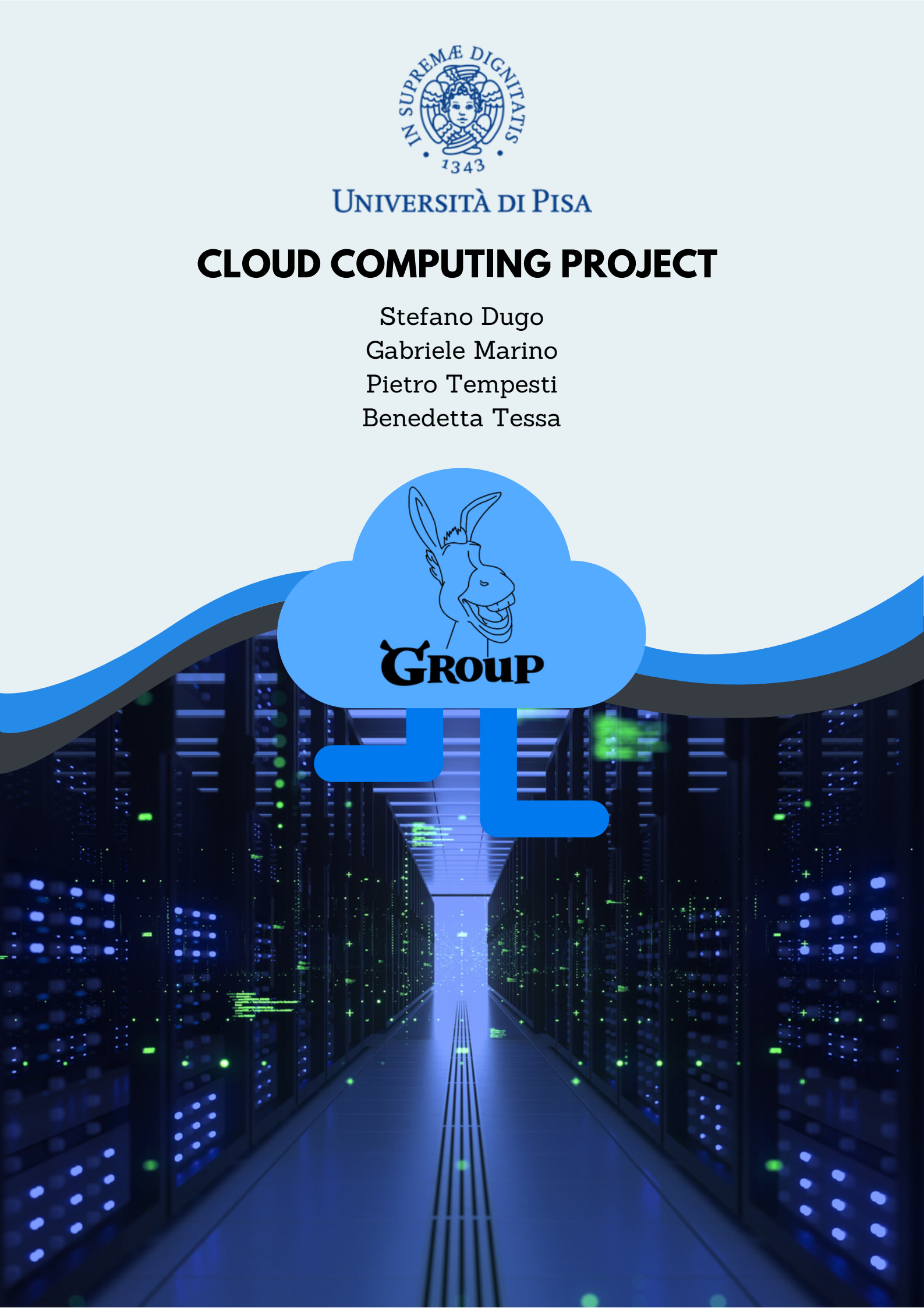

CLOUD COMPUTING PROJECT

Stefano Dugo
Gabriele Marino
Pietro Tempesti
Benedetta Tessa

1 Parameters

To build a Bloom filter, we need 4 parameters:

- *p*: false positive rate
- *k*: number of hash functions
- *m*: number of bits of the bloom filter
- *n*: *number of keys in input to the bloom filter*

They are all dependent from each other by means of mathematical relations, so once we fix a parameter, we can easily compute the others.

Here are such relations:

$$m = -n\frac{\ln p}{\ln 2^{2}}$$

$$k = \frac{m}{n}\cdot \ln 2$$

$$p \approx \left(1 - e^{-\frac{kn}{m}}\right)^{k}$$

1.2 Computing of parameters

In the bloom filter application, p is a user-defined parameter and with further analysis on the equations above we can observe that $k = \left\lceil -\frac{\ln(p)}{\ln(2)} \right\rceil$ so we used this formula to compute *k*.

The values of *n* and *m*, however, depend on the input dataset.

Each line of such dataset is represented by the following tuple:

< movie id, average rating, number of votes>

We have a bloom filter for each possible rating, an integer between 1 and 10. Each filter has an id associated, equal to the value of the rating it represents.

The rating for a movie is obtained by rounding to the closest integer the average rating.

What we had to do was count the number of movies for each possible rating to get the value of *n* for each filter.

Computing the values of *m* is now trivial, using the prior formula.

Algorithm 1 MAPPER

Require: *DOC*, the dataset from which we want to count the entries

```
for each rating ∈ DOC do
    if word == rating then
        rating ← rating.round()
        emit(rating, 1)
    end if
end for
```

Algorithm 2 REDUCER

Require: *INPUT_LIST*, a list with a rating and a series of ones
Require: *p*, false positive rate desired

```
Initialize n ← 0
Initialize m ← 0
for each value ∈ INPUT_LIST do
    n ←n+value
end for
m ← -(n * ln(p))/(ln(2)^2)

write_to_dfs(m)
```

These are the values of m obtained using $p = 0.01$, resulting in a value of $k = 7$

Rating	Parameter M
1	24462
2	63645
3	170777
4	417698
5	981999
6	2104851
7	3557984
8	3392795
9	1085029
10	154550

2 Implementation

To build the filters, we had to perform the following steps:

1. Get the *movie id* and the *rounded average* rating from the input file
2. Compute *k* different hash values having the *movie id* as key
3. Initialize each filter with the corresponding *m* as an array of Booleans set to *false*
4. Set to *true* the positions in the array identified by the hash values

Algorithm 3 MAPPER

Require: $HASH\_FUNCTIONS$, list of hash functions
Require: *input_split*, record in input to the Mapper
Require: k, the number of hash functions

Initialize *movie_name* ←input_split.movie_name
Initialize *avg_rating* ←round_next_int(input_split.avg_rating)
Initialize $hash\_values = newARRAY(k)$

for each $hash\_function \in HASH\_FUNCTIONS$ **do**
  $hash\_values[pos]$ ←hash_function(*movie_name*)
end for

emit(avg_rating, hash_values)

Algorithm 4 REDUCER

Require: $INPUT\_LIST$, list of hash results computed in the Mapper
Require: *key*, average rating for a movie
Require: m, dimension of the bloom filter

Initialize *Bloom_Filter* ←new ARRAY(m)
Bloom_Filter[i] ← 0 for each i = 0,...,m-1

for each $HASH\_ARRAY \in INPUT\_LIST$ **do**
  for each $value \in HASH\_ARRAY$ **do**
    Bloom_Filter[*value*] ← 1
  end for
end for

write_to_dfs(*Bloom_Filter*)

Let's see how it was implemented in first in Spark and then in Hadoop:

2.1 Spark Implementation Details

2.1.1 Initial setup

In order to run on the cluster our application we have to pass the same python environment on all the cluster machines, because the mmh3 python package, used to compute the *MurmurHash* functions, is not built in.

These are the steps to create a virtual environment with all the needed packages:

1. Only on *debian/ubuntu* install python venv: *sudo apt-get install python3-venv*
2. create a virtual environment and activate it
   a. *python -m venv pyspark_venv*
   b. *source pyspark_venv/bin/activate*
3. install all the needed dependencies using *pip*: *pip3 install pyspark mmh3 venv-pack*
4. zip the virtual environment in a *.tar.gz* archive: *venv-pack -o pyspark_venv.tar.gz*

After these steps, we must pass the archive to the cluster in the *spark-submit* command:

export PYSPARK_PYTHON=./environment/bin/python

spark-submit --archives pyspark_venv.tar.gz#environment main.py [input path] [output path] [false positive rate]

Inside the script there's a command used to set the virtual environment uploaded on the dfs as the default environment for the execution of the application.

2.1.2 Main steps

These are the main steps of the algorithm implemented in spark, given as input a value for *p*, the input dataset and the output path:

1. compute the value of *k* given *p*
2. import the dataset
3. compute the value of *m* for each possible vote given *p* and the dataset
4. compute the hash functions for each entry of the dataset
5. map these results using the average votes as a key
6. aggregate all the hash results for each vote
7. build the bloom filters using the aggregated positions
8. export the bloom filters

2.1.1 Partitioning and Broadcast Variables

We splitted the initial dataset in 4 different partitions, one for each node in our cluster, to ensure better performances and lessen the execution time.

We also decided to cache the partitioned dataset and broadcast *p, k, m* and *n* since they need to be used by all workers.

For this reason, it could be useful to ship those values only once to the working nodes to limit network overhead and reduce communication costs.

2.2 Hadoop Implementation Details

2.2.1 Splits and Configurations

The input was splitted using *NLineInputFormat*, dividing the number of lines of the dataset by 4, giving an equal amount of input lines to all the nodes in our cluster; we implemented a method *getLines* to obtain the total number of lines of the input dataset, which has been rounded to the next closer integer value (using *Math.ceil()*).

2.2.2 Bloom Filter Construction in Hadoop

For implementing the bloom filter construction, we first need to retrieve the values of *m* from *HDFS*. For this purpose, we wrote a *readM* method, which retrieves for each key the corresponding value of *m*, and we put these values, as well as the value of *k*, in the map reduce job configuration. The hash values are computed using the *MurmurHash* functions' family; since we got an Integer overflow for some hash values, and so some of them were negative even after doing the modulo operation, we needed to compute each value as follows:

$$(hashValue \,\%\, m + m) \,\%\, m$$

For giving the array in output to the mapper, we needed to implement a class *IntArrayWritable* to implement the *ArrayWritable* class of Hadoop, for serializing an array of integers.

The output value of the reducer is an array of boolean, so we could spare as much space as possible; for giving it in output, we implemented another class which extends *ArrayWritable*, which is *BooleanArrayWritable*.

3 Experimental Results

Using the same dataset and the same *p* given in input to the construction phase, we tested the correctness of our algorithm by calculating the false positive rate for each bloom filter and comparing it with *p*.

For each tuple of the dataset we extracted the *movie_id* and, rounded average rating and computed the hash values with the same hash functions used prior.

Afterwards, for each filter we checked whether that given *movie_id* was present or not so that we could calculate the number true negatives and false positives.

Presence Check:

If at least one of the values contained in the positions identified by the hash values is *false,* then we can say the movie is not present for sure (we can increment the number of true negatives).

On other hand, if all values are *true* this means the movie could be present and the bloom filter outputs *maybe*.

The number of false positives is incremented each time we get *maybe* as an answer, but the rating associated with that movie is different from the *id* of the bloom filter.

The false positive rate can be easily computed as

$$\textit{False positive rate} = \frac{false\ positives}{false\ positives + true\ negatives}$$

Spark:

Rating		P = 0.01	P = 0.05	P = 0.1
1	False positives:	12481	63996	127436
	False positive rate:	0.01003	0.05142	0.10239
2	False positives:	12872	63894	126979
	False positive rate:	0.01038	0.05155	0.10245
3	False positives:	12228	63368	125850
	False positive rate:	0.00994	0.05150	0.10228
4	False positives:	11999	61824	122260
	False positive rate:	0.01002	0.05164	0.10212
5	False positives:	11349	58708	117782
	False positive rate:	0.00986	0.05101	0.10233
6	False positives:	9953	50974	102275
	False positive rate:	0.00999	0.05101	0.10235
7	False positives:	9192	45995	92095
	False positive rate:	0.01017	0.05087	0.10186
8	False positives:	8837	44556	89683
	False positive rate:	0.01012	0.05100	0.10266
9	False positives:	11843	59525	118073
	False positive rate:	0.01027	0.05160	0.10236
10	False positives:	12337	62889	126369
	False positive rate:	0.01002	0.05109	0.10265

Hadoop:

Rating		P = 0.01	P = 0.05	P = 0.10
1	False Positives:	12905	63269	127694
	False Positive Rate:	0.01037	0.05084	0.10260
2	False Positives:	12669	63901	126835
	False Positive Rate:	0.01021	0.05151	0.10225
3	False Positives:	12165	62517	125409
	False Positive Rate:	0.00990	0.05086	0.10202
4	False Positives:	12056	62117	123190
	False Positive Rate:	0.01002	0.05161	0.10236
5	False Positives:	11536	58447	117072
	False Positive Rate:	0.01008	0.05106	0.10228
6	False Positives:	10552	52080	105371
	False Positive Rate:	0.01027	0.05068	0.10254
7	False Positives:	8754	44319	90069
	False Positive Rate:	0.00999	0.05060	0.10283
8	False Positives:	8832	45956	91423
	False Positive Rate:	0.00989	0.05145	0.10235
9	False Positives:	11305	58020	117098
	False Positive Rate:	0.00997	0.05117	0.10327
10	False Positives:	12287	63951	125680
	False Positive Rate:	0.00999	0.05195	0.10210