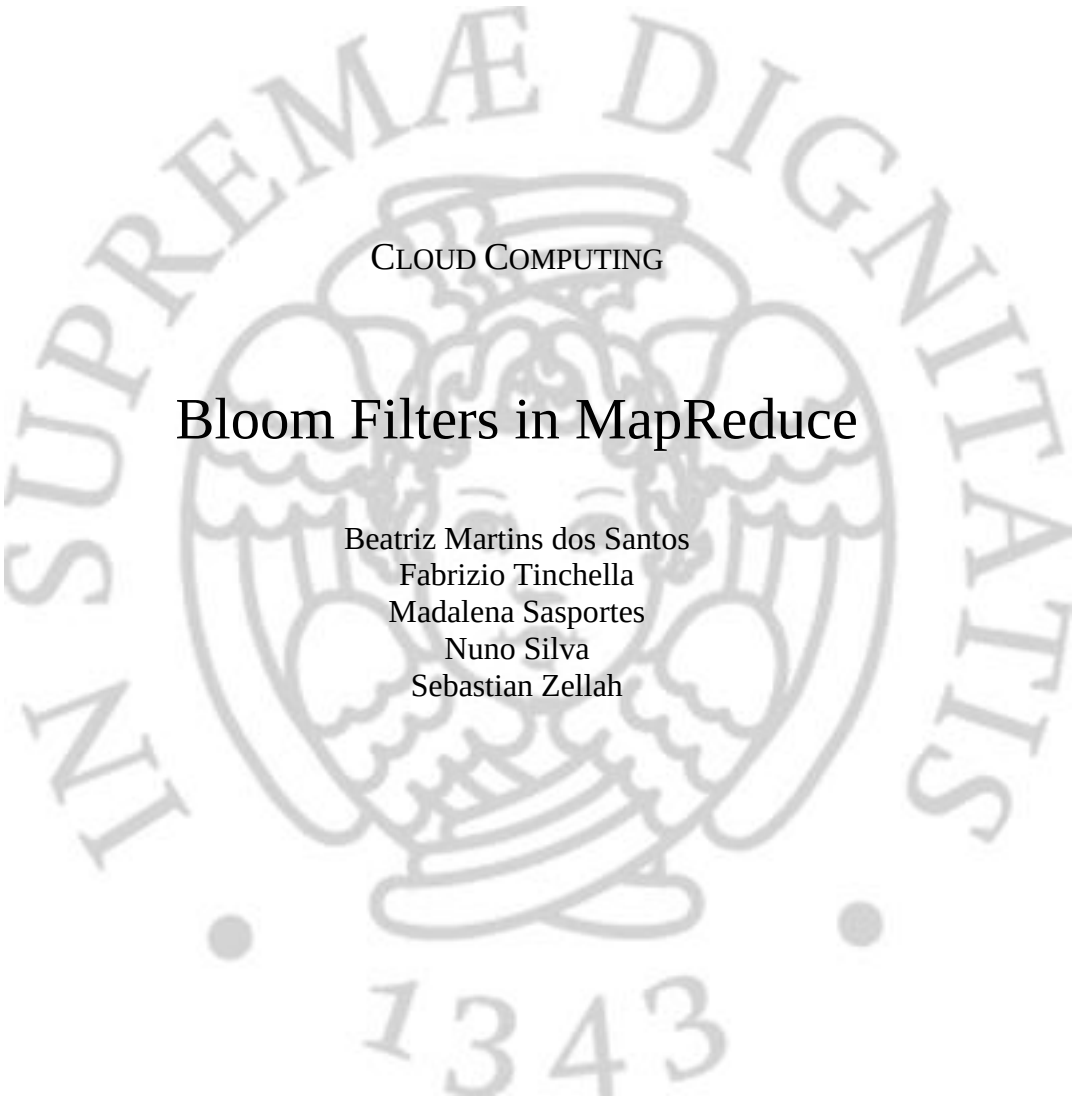
UNIVERSITÀ DI PISA
MASTER PROGRAM IN COMPUTER SCIENCE



CLOUD COMPUTING

Bloom Filters in MapReduce

Beatriz Martins dos Santos
Fabrizio Tinchella
Madalena Sasportes
Nuno Silva
Sebastian Zellah

Professor Nicola Tonellotto
a.a. 2021/2022

Bloom Filters in MapReduce

1. Description of the problem faced

The goal for this project was to build a bloom filter over the ratings of movies listed in the IMDb datasets. Computing a bloom filter for each rating value.

2. Design

The design that we used for implementing the construction of the bloom filter using the Hadoop framework and the Spark framework differs from each other. We did a different design in order to take advantage of both frameworks' features.

- Design for the Hadoop Framework: (pseudocode)

```
class MAPPER

    method setUp()
        filters = new ArrayList<BloomFilter>(11)
        for pos p in filters do
          set p to null


    method MAP(split s)
        for all line l in split s do
            rating = Math.round(l.getRating())

            if filters.get(rating) is equal to null
                BloomFilter bf = new BloomFilter
                bf.add(new Key(idInBytes))
                filters.add(rating, bf)
            else
                filters.get(rating).add(new Key(idInBytes))
            endif

    method cleanup()
        for all rating r in filters do
            if filters.get(r) not null
                EMIT(r, filters.get(r))
            endif
        clear filters

class REDUCER
    //initialize BloomFilter result


    method REDUCE(int rating, Iterable bloomFiltersList)
        result = new BloomFilter
        for all values v in bloomFiltersList do
            result = result OR v
```

```
        write bloom filter to file system
        EMIT(int rating, Text result)
```

- Design for the Spark Framework: (pseudocode)

```
class BLOOM_FILTER_SPARK

    RDD moviesInGroupSorted => tuple(rating, List(movieIds)))

    method bloom_construction(RDD moviesInGroupSorted)
        bloom = new BloomFilter
        for movie id in moviesInGroupSorted[1] do
            bloom.fillUp(moviesInGroupSorted[1][id])
        return bloom
```

3. Implementation using Hadoop Framework

In the Hadoop implementation each mapper will fetch the corresponding m and k from the job configuration and also build their own array to save each ratings' bloom filter. After that, each mapper processes one split and each split has the number of lines that we define as parameters as running the program. The map function runs once for each record in that split. A record corresponds to one line of the file. Each record is split to get the roundup of the rating and the movie id.

To deal with the first line of the input file (header) we decided to implement a try/catch block that tries to round the rating and if cannot do it, raises an exception and continues to the following lines of the file.

If there isn't a bloom filter already created for that rating (indexed position), a new bloom filter will be created (with the corresponding m and k) and the movie id will be hashed, with the murmur hash function, and added to that same filter. On the other hand, if the filter already exists for that rating, the movie id is hashed and then added to the filter.
After processing the split, each mapper will do the cleanup process and write their bloom filters to the hdfs.

The reducer will declare a new bloom filter *result* (used as the final bloom filter for each rating). On the setup it'll fetch the variables m and k that were given in the job configuration.
Then it proceeds to initialize the result to a new bloom filter and for each key (= rating) it will iterate all the bloom filters and merge them into the variable *result*. We decided to implement this last step with an OR function.

Result will then be written to the file system, in the corresponding directory for that key (rating). It will also be written to the hdfs as a pair (key, Text) converting the result into a string.

4. Implementation using Spark Framework

In our Spark implementation, we fetch the data file and use a filter to remove the header, establishing the initial RDD to apply the transformations to.
Each row will then be turned into a tuple (Rating, ID), using the map function to the previous RDD. This will permit to sort and group the movies by rating and form a list of movies ids (RDD *movies_in_groups_sorted*) .

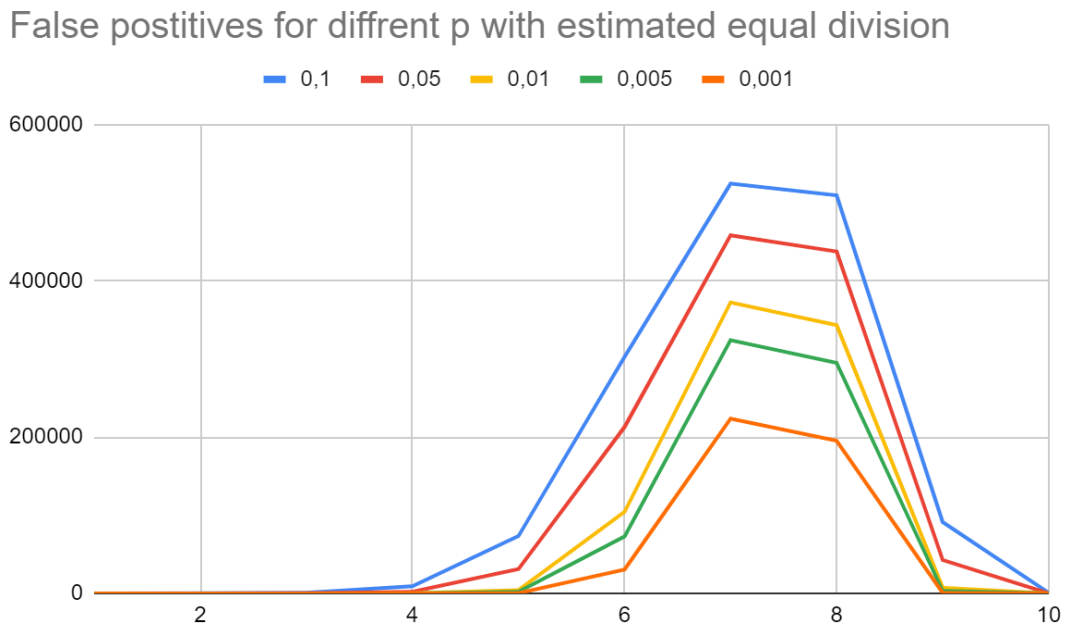
It's to this RDD that will be applied the transformation with the function *bloom_construction()*. This function receives each tuple of the dataset (*Rating, List(MoviesId)*) and builds the bloom filter for each rating. We build a bloom filter class to use the Murmur Hash Function from python libraries (mmh3). This function initializes a new bloom filter, takes the list of *moviesIds* from that rating, iterates it and hashes each *movieId* to then be added to the bloom filter. The bloom filter is then returned and after the transformation is finished, we'll obtain a new RDD that keeps tuples in the form of (*Rating, BloomFilter*).

5. Experimental Results

So first we are going to describe the results for the implementation with Hadoop. Second, we describe the results using the spark implementation.

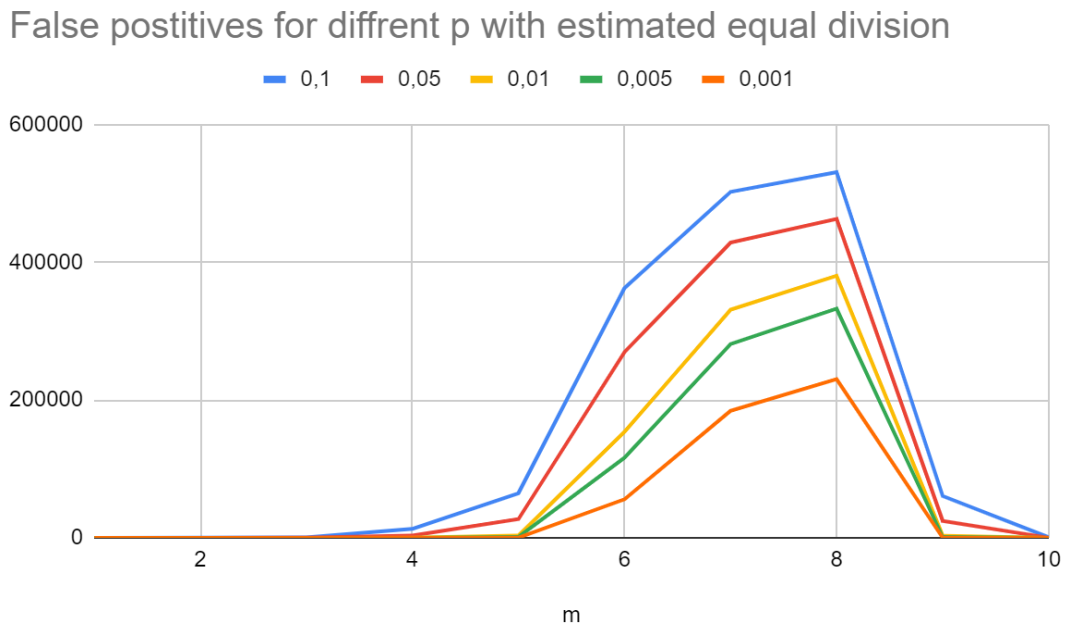
False positives assuming equal amount of movies for each rating in hadoop

	1	2	3	4	5	6	7	8	9	10
h - 0,1	3	42	733	9079	73301	302733	524435	509401	91214	590
h - 0,05	0	2	71	1896	30981	212810	458126	437328	42783	60
h - 0,01	0	0	0	44	4075	104225	372264	343160	6993	0
h - 0,005	0	0	0	8	1817	72611	324010	295032	3083	0
h - 0,001	0	0	0	0	250	30574	223639	195261	587	0

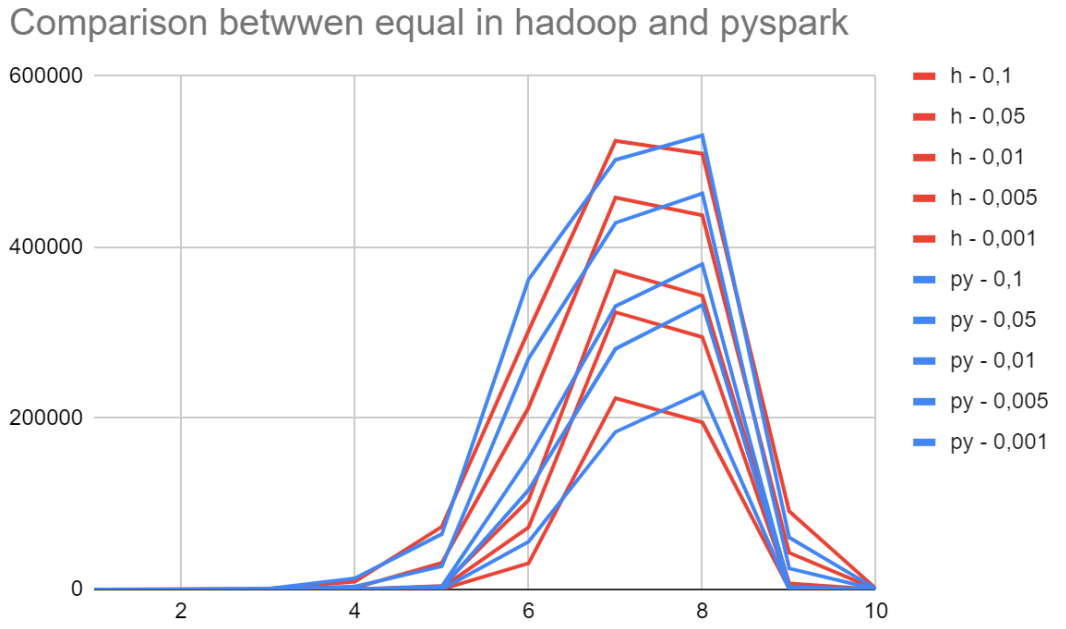


False positives assuming equal amount of movies for each rating in pyspark

	1	2	3	4	5	6	7	8	9	10
py - 0,1	3	76	612	12940	64669	362421	502160	530874	60680	568
py - 0,05	0	2	60	3302	27067	269854	428643	462990	24595	64
py - 0,01	0	0	0	75	3195	153942	331122	380369	2843	0
py - 0,005	0	0	0	23	1353	116268	281317	332794	1120	0
py - 0,001	0	0	0	1	183	55923	184146	230386	160	0



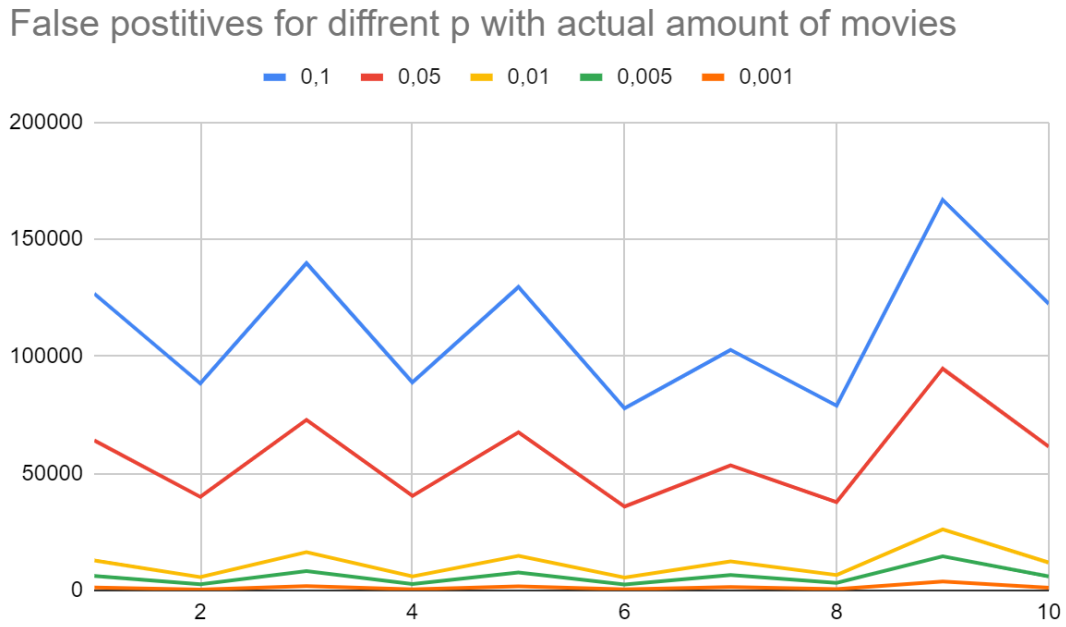
graph showcasing both implementations



False positives assuming actual specific amount of movies for each rating in hadoop

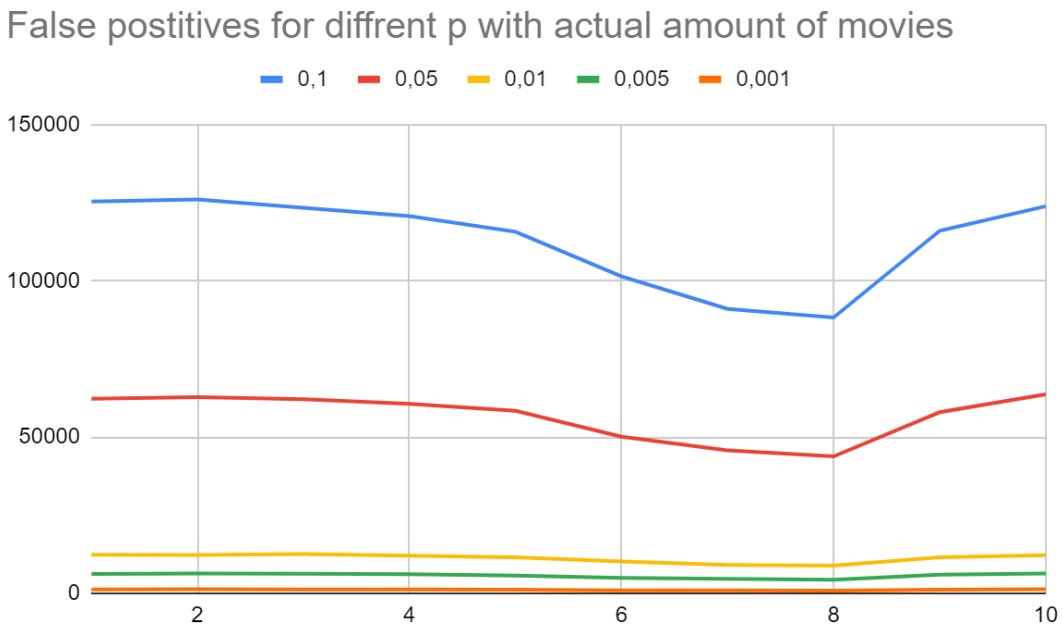
	1	2	3	4	5	6	7	8	9	10
h - 0,1	126773	88440	139863	88906	129641	77743	102777	78967	166857	122504
h - 0,05	64100	39991	72862	40363	67554	35838	53399	37737	94724	61438

h - 0,01	12761	5668	16291	6002	14746	5495	12365	6606	26046	11897
h - 0,005	6188	2582	8287	2718	7673	2523	6545	3215	14590	6010
h - 0,001	1251	391	1856	423	1699	439	1441	573	3857	1186

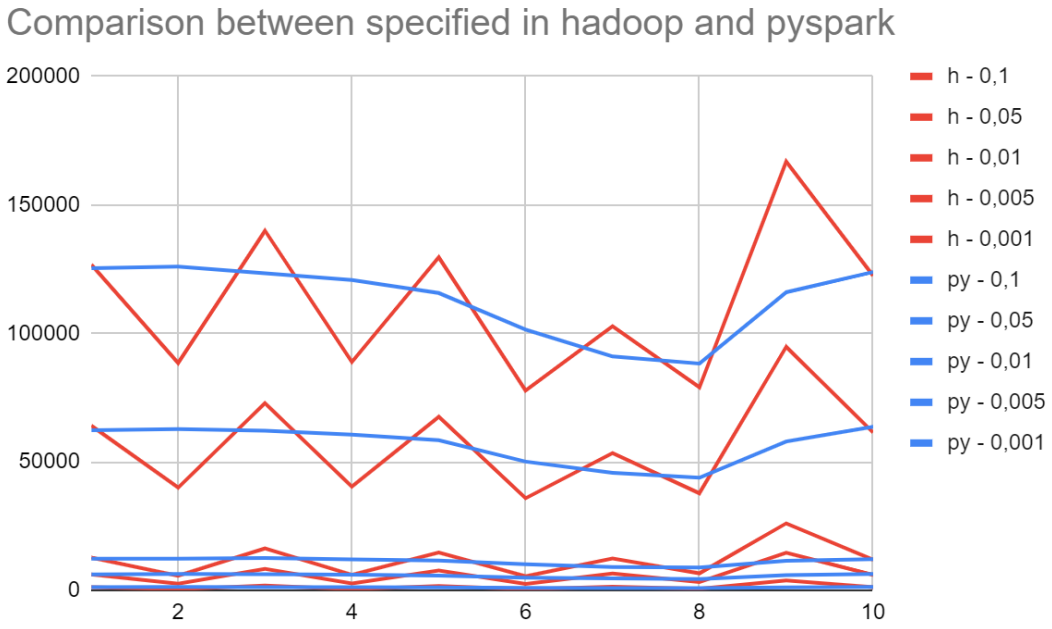


False positives assuming actual specific amount of movies for each rating in pyspark

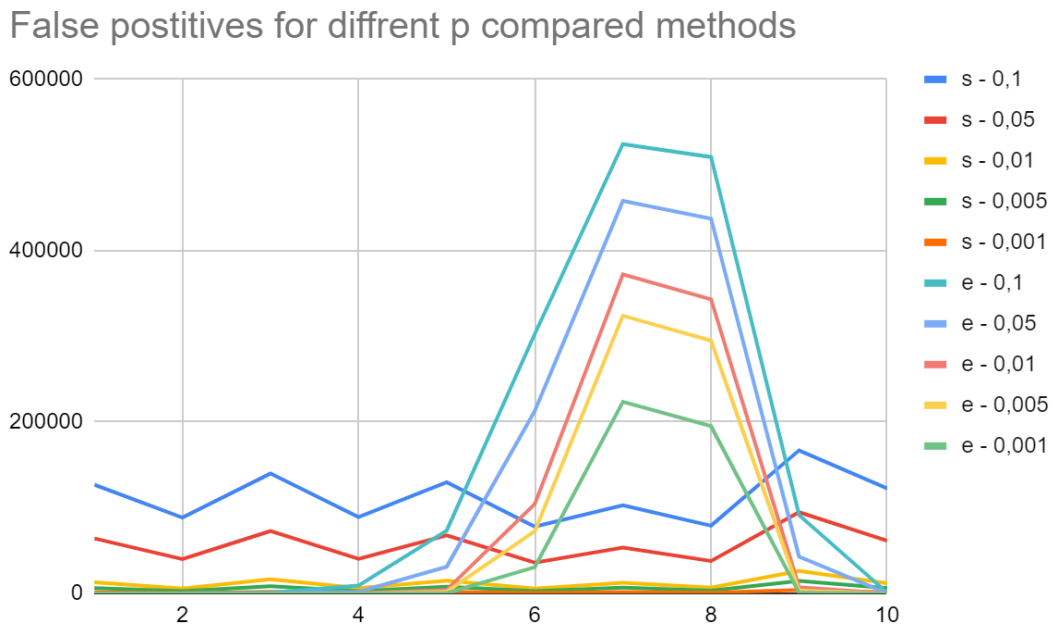
	1	2	3	4	5	6	7	8	9	10
py - 0,1	125365	126016	123362	120723	115702	101405	91001	88264	116008	123850
py - 0,05	62266	62781	62121	60593	58418	50094	45717	43796	57947	63639
py - 0,01	12348	12272	12557	12044	11551	10158	9050	8876	11530	12159
py - 0,005	6211	6339	6264	6100	5687	4961	4604	4343	5933	6358
py - 0,001	1232	1306	1246	1235	1150	953	909	885	1165	1276



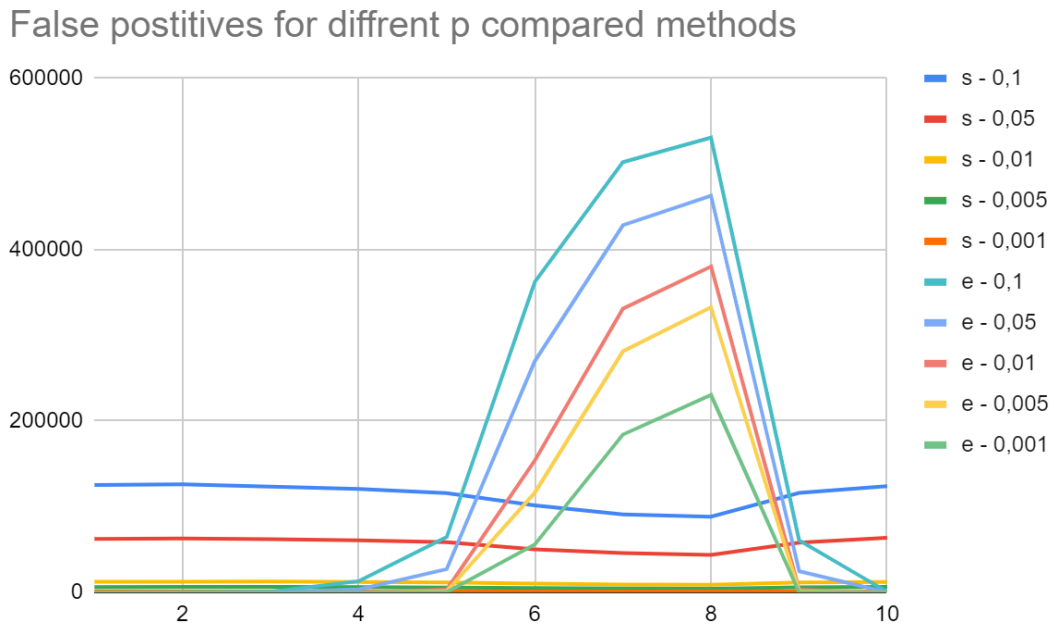
graph showcasing both implementations



Comparison between equal division and specific division in hadoop



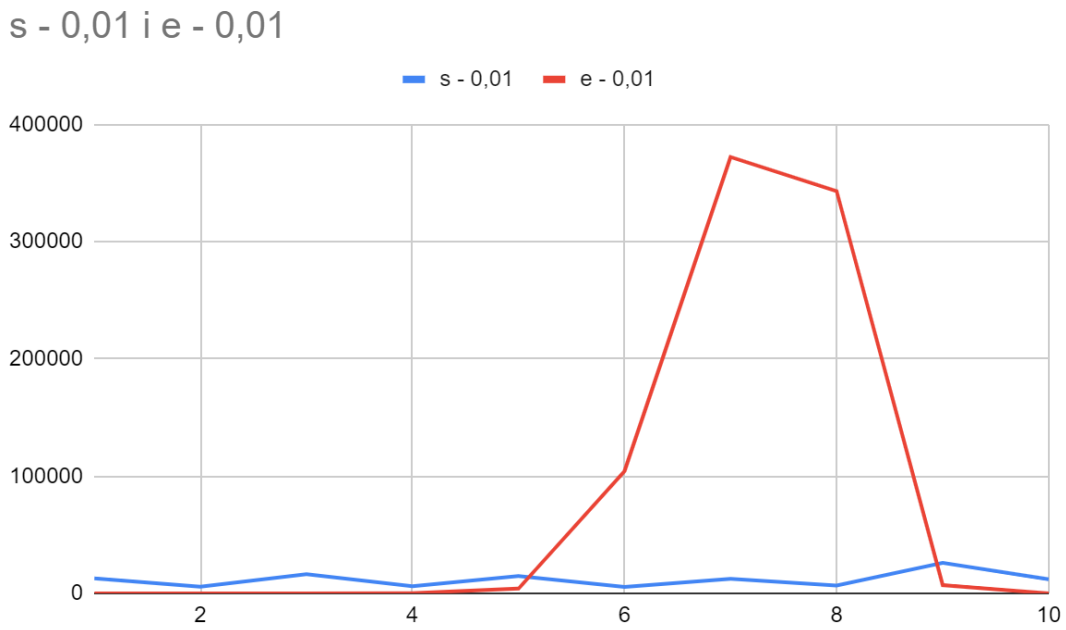
Comparison between equal division and specific division in pyspark



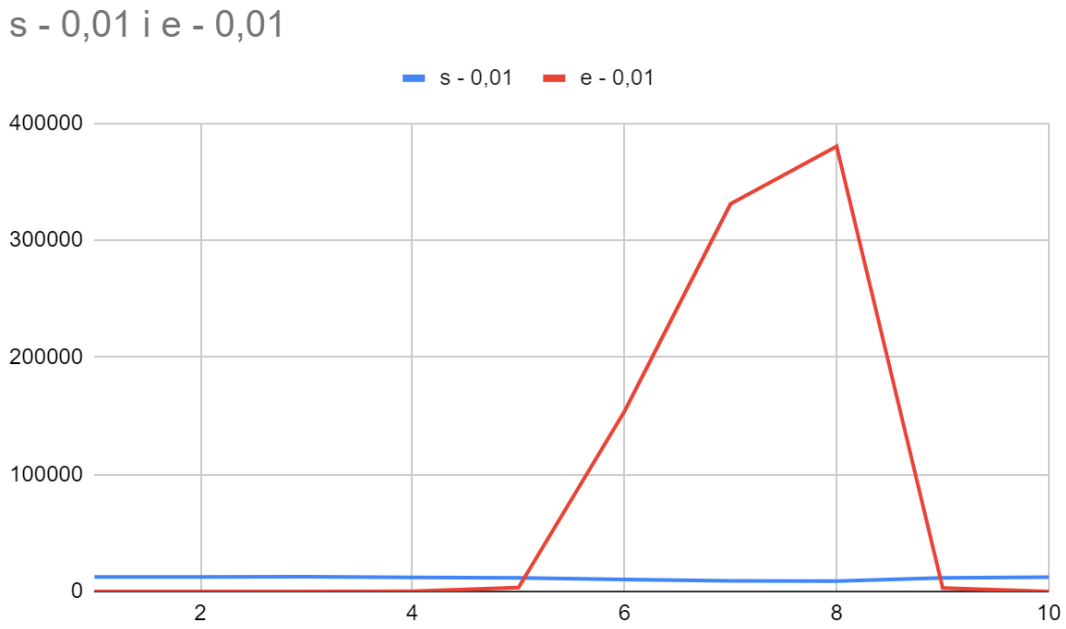
Equal division resulted in much bigger filter sizes that needed in less popular ratings, but too small filter sizes in more popular ratings. Therefore in case of popular ratings, filters were filled up and resulted in being much less useful, while for less popular ratings they were occupying too much space.

For specific division filter sizes where consistent and with the p even as big as 0,01 they were getting very satisfying results

hadoop


s - 0,01 i e - 0,01
s - 0,01
e - 0,01
400000
300000
200000
100000
0
2
4
6
8
10

pyspark


s - 0,01 i e - 0,01
s - 0,01
e - 0,01
400000
300000
200000
100000
0
2
4
6
8
10