University of Pisa
School of Engineering

Cloud Computing

A MapReduce implementation of Bloom filters

Supervisors
Nicola Tonellotto
Carlo Vallati

Students
Marco Pinna
Olgerti Xhanej
Federico Cristofani
Matteo Biondi

July 5, 2022

Contents

1 Introduction **2**

2 Overview and design choices **4**
2.1 Bloom filters 4
2.2 Algorithms design 5
2.2.1 Compute parameters 5
2.2.2 Create Bloom filters 6
2.2.3 Test filters 9
2.3 Considerations and hypotheses 10

3 Hadoop **11**

4 Spark **14**

5 Performance **16**
5.1 Hadoop map output bytes 16
5.2 Hadoop wall time 19
5.3 Hadoop vs Spark 21

Chapter 1

Introduction

This work concerns the design, implementation and testing of Bloom filters using the MapReduce framework, both in Java and in Python.
The specifications are detailed in the following:

> You will build a bloom filter over the ratings of movies listed in the IMDb datasets.
> The average ratings are rounded to the closest integer value, and you will compute a bloom filter for each rating value.
> In your Hadoop implementation, you must use the following classes:
>
> - `org.apache.hadoop.mapreduce.lib.input.NLineInputFormat`: splits N lines of input as one split;
> - `org.apache.hadoop.util.hash.Hash.MURMUR_HASH`: the hash function family to use.
>
> In your Spark implementation, you must use/implement analogous classes.
> In this project you must:
>
> 1. design a MapReduce algorithm (using pseudocode) to implement the bloom filter construction;
> 2. implement the MapReduce bloom filter construction algorithm using the Hadoop framework;
> 3. implement the MapReduce bloom filter construction algorithm using the Spark framework;
> 4. test both implementations on the IMDb ratings dataset, computing the exact number of false positives for each rating;
> 5. write a project report detailing your design, implementations and reporting the experimental results.

The work is organized as follows:

- In chapter 2 a brief overview of Bloom filters is given. Then the general algorithm is presented, with two possible implementations, together with design choices and hypotheses or considerations that have been made about the dataset to be used or the use-cases of the Bloom filters.

- Chapter 3 concerns the Java implementation of the algorithm, that makes use of Hadoop framework.
- Chapter 4 concerns the Python implementation of the algorithm, that makes use of Spark framework
- In chapter 5 an evaluation of the performance of both implementations is provided.

The entire codebase is available at
`https://github.com/MPinna/MapReduceBloomFilter` .

Chapter 2

Overview and design choices

In this chapter we firstly present a brief overview of what Bloom filters are (partly taken from the project specifications); then we show the general algorithm, with two possible implementations along with their respective pseudo-code, together with design choices and hypotheses that have been made during the design process.
Finally, some considerations about the dataset to be used and the use cases of the Bloom filter are made.

2.1 Bloom filters

A *Bloom filter* is a space-efficient probabilistic data structure that is used for membership testing.
A Bloom filter consists of a bit-vector with `m` elements and `k` hash functions to map `n` keys to the m elements of the bit-vector.
The two possible operations that can be done on a Bloom filter are `add` and `test`.
Given a key id_i, every hash function $h_1, ..., h_k$ computes the corresponding output positions and sets the bit in that position to 1.
The space efficiency of Bloom filters comes at the cost of having a non-zero probability of false positives. The false positive rate (FPR) of a Bloom filter is denoted by `p`.
Therefore the two possible outcomes of the `test` function of the Bloom filter are “Possibly in set” or “Definitely not in set”.

The relations between n, m, k and p are expressed by the following formulas, which can be used to compute the optimal value the Bloom filters parameters:

$$m = -\frac{n \ln p}{(\ln 2)^2}, \qquad k = \frac{m}{n} \ln 2, \qquad p \approx (1 - e^{-\frac{kn}{m}})^k \tag{2.1}$$

In this use case, Bloom filters will be used to check whether a movie in the IMDb dataset belongs to the set of movies having a certain average rating.
Movies in the IMDb datasets can be rated by users from 1.0 to 10.0. Rounding the

rating to the nearest integer yields 10 possible ratings. A total of 10 Bloom filters will then be built.

A total of three phases were designed for this purpose, each of which is carried out using MapReduce:

- **computation of the best parameters** for the filter, (possibly with constraints on the value of k): the distribution of movies among the different ratings is not uniform, therefore a different m for each of the 10 Bloom filters can be computed, according to how many movie that filter is going to contain;

- **creation of the Bloom filters**

- **testing of the Bloom filters**: the Bloom filters created in the previous step are designed with a certain FPR p. The testing is used to ensure that the filters do in fact have that FPR.

2.2 Algorithms design

2.2.1 Compute parameters

This initial phase takes care of counting how many movies belong to each of the 10 rating buckets and computing the optimal parameters to properly size the filters in the next phase; this is done to ensure that the filters have a certain FPR p without oversizing the bitArray, which would result in a waste of resources.

Algorithm 1 Compute Parameters Mapper

```
procedure ComputeParamsMapper(splitId a, split s)
    ratingCount ← new int[NUM_RATINGS]
    for movie m in split s do
        ratingCount[m.rating - 1] ← rating_count[m.rating - 1] + 1
    end for
    for i=1,2,...,MAX_RATING do
        emit(i, ratingCount[i-1])
    end for
end procedure
```

Algorithm 2 Compute Parameters Reducer

```
procedure ComputeParamsReducer(rating r, ratinCounts [c1, c2, ..., c_10])
    ratingCountSum ← 0
    for ratingCount c in ratingCounts do
        ratingCountSum ← ratingCountSum + c
    end for
    if no constraints on K then                ▷ passed as input to the script
        m ← computeBestM(ratingCountSum, p)
        k ← computeBestK(ratingCountSum, m)
    else
        k ← constrainedK
        m ← computeBestM(ratingCountSum, p, k)
    end if
    emit(r, (p, ratingCountSum, m, k))
end procedure
```

2.2.2 Create Bloom filters

The general idea of this MapReduce implementation is to split the IMDb dataset into partitions and have each mapper compute, from the movies contained in one partition, part of the information needed to build the relative final Bloom filters. The reducer(s) will then combine the data received from the mappers and merge it into a total of 10 Bloom filters.
Two different algorithms have been designed for the task: the first one computes the indexes on the mappers and sends them to the reducers which takes care of creating the Bloom filters; the second one creates the Bloom filters on the mapper(s), fills them partially and lets the reducers merge all the filters into the final 10 Bloom filters.
Let us see these two implementations more in detail:

With Indexes

Algorithm 3 Mapper

```
procedure MapperWithIndexes(splitId a, split s)
    for every movie m in split s do
        rating ← round(m.rating)
        len ← getBitArrayLen(rating)
        bitArrayIndexes ← new Array[k]
        for i=1,2,...,k do
            bitArrayIndexes[i] ← h_i(m.id) % len
            emit(i, bitArrayIndexes)
        end for
    end for
end procedure
```

Algorithm 4 Reducer

```
procedure ReducerWithIndexes(rating r, bitIndexes[b1[], b2[],..., bj[]])
    len ← getBitArrayLen(rating)
    bloomFilter ← new BitArray[len]
    bloomFilter.set(allZeros)
    for every bitIndex b in bitIndexes do
        for every index i in b do
            bloomFilter[i] ← 1
        end for
    end for
    emit(r, bloomFilter)
end procedure
```

With Bloom Filters

Algorithm 5 Mapper

```
procedure MapperWithBloomFilters(splitId a, split s)
    for i=1,2,...,MAX_RATING do                ▷ Create 10 empty Bloom filters
        len ← getBitArrayLen(i)
        bloomFilter_i ← new BitArray[len]
        bloomFilter_i.set(allZeros)
    end for
    for every movie m in splits do
        rating ← round(m.rating)
        bloomFilter_i.add(m.id)
    end for
    for i=1,2,...,MAX_RATING do
        emit(i, bloomFilter_i)
    end for
end procedure
```

Algorithm 6 Reducer

```
procedure ReducerWithBloomFilters(rating r, bloomFilters [bf1, bf2, ...,
bfj])
    len ← getBloomFilterLen(r)
    bloomFilterResult ← new BitArray[len]
    bloomFilterResult.set(allZeros)
    for every bloomFilter bf in bloomFilters do
        bloomFilterResult ← bitwiseOr(bloomFilterResult, bf)
    end for
    emit(r, bloomFilterResult)
end procedure
```

2.2.3 Test filters

Algorithm 7 Testing mapper

```
procedure TestMapper(splitId a, split s)
    savedBloomFilters ← loadBloomFiltersFromHDFS()
    trueNegativeCount ← new int[NUM_OF_RATINGS]
    falsePositiveCount ← new int[NUM_OF_RATINGS]
    for every movie m in split s do
        movieRating ← round(m.rating)
        for currentRating in 1,2,...,MAX_RATING do
            bloomFilter ← savedBloomFilters[currentRating]
            testResult = bloomFilter.test(m.id)
            if testResult == true and movieRating ≠ currentRating then
                falsePositiveCount ← falsePositiveCount + 1
            end if
            if testResult == false and movieRating ≠ currentRating then
                trueNegativeCount ← trueNegativeCount + 1
            end if
        end for
    end for
    counter← new int[2]
    for every bloomFilter in savedBloomFilters do
        bloomFilterRating ← bloomFilter.getRating()
        counter[0] ← falsePositiveCount[bloomFilterRating - 1]
        counter[1] ← trueNegativeCount[bloomFilterRating - 1]
        emit(bloomFilterRating, counter)
    end for
end procedure
```

Algorithm 8 Testing reducer

```
procedure TestReducer(rating r, counters [c1[], c2[], ..., cj[]])
    falsePositiveCounter = 0
    trueNegativeCounter = 0
    for counter c in counters do
        if c[0] ≥ 0 and c[1] ≥ 0 then
            falsePositiveCounter ← falsePositiveCounter + c[0]
            trueNegativeCounter ← trueNegativeCounter + c[1]
        end if
    end for
    if falsePositiveCounter + trueNegativeCounter > 0 then
        FPR ← falsePositiveCounter/(falsePositiveCounter + trueNegativeCounter)
        emit(r, FPR)
    end if
end procedure
```

2.3 Considerations and hypotheses

The IMDb dataset consists of a .tsv file with approximately 1200000 movies, one per row.
Although the IMDb dataset is updated daily, it was assumed that the Bloom filters are to be used with a fixed dataset.

Chapter 3

Hadoop

The first version was implemented in Java, using the Hadoop framework.
The directory structure of the project is the following:

```
hadoop
└── main/java/it/unipi/hadoop
    ├── BloomFilter.java
    ├── MapRedComputeParams.java
    ├── MapRedBloomFilter.java
    ├── MapRedFalsePositiveRate.java
    └── Util.java
```

- The `BloomFilter.java` class contains the implementation the Bloom filter.
  The *bitArray* structure of the Bloom filter was realized by using the `BitSet` class available in the `java.util.BitSet` Java library. It already comes with built-in methods for setting/resetting all the bits, performing bit-wise operations on it and serializing/deserializing the array. The family of hash functions that was used — as requested in the specifications — is MurmurHash, which can be found in the `org.apache.hadoop.util.hash.MurmurHash` Java library.

- The `MapRedComputeParams.java` class is used to compute the optimal filter parameters according to the constraints passed as input via command line.

- The `MapRedBloomFilter.java` is the main class that contains all the MapReduce logic for the creation of the Bloom filters. It contains both the versions of the algorithm: the one *withIndexes* and the one *withBloomFilters*. One can toggle between the two by simply changing one command line argument when launching the program on the cluster.

- The `MapRedFalsePositiveRate.java` is used to test the Bloom filters created by the previous class and check if the empirical FPR is actually consistent with the theoretical one that was yielded by `MapRedComputeParams.java` and used by `MapRedBloomFilter.java`.

- The `Util.java` class contains constants and utility functions that are used in the rest of the code, such as the ones for parsing and splitting the input rows.

In the implementation of all the MapReduce classes, `setup()` and `cleanup()` methods were used: `setup()` to fetch the necessary parameters and initialize the data structures and `cleanup()` to emit the data collected during the previous elaborations.
All the `MapRed*` classes take, as command line argument — among the other parameters, the size of the split to be given as input to each mappers (i.e. the `num_lines_per_split` argument); this parameter directly controls the number of mappers that are instantiated during the Map phase. To do this, the `NLineInputFormat` class was used, as required by the specifications, which can be found in the `org.apache.hadoop.mapreduce.lib.input.NLineInputFormat` Java library.

Usage

The three `MapRed*` classes can be run with the following commands:

```
hadoop jar it/unipi/hadoop/BloomFilter/1.0/BloomFilter-1.0.jar it.unipi.
    hadoop.MapRedComputeParams title.ratings.tsv output 0.01 157000
```

`title.ratings.tsv` is the input file, `output` is the name of the output file, `0.01` is the desired value for the FPR p and `157000` is the size of the partition to be handled to each mapper.

```
hadoop jar it/unipi/hadoop/BloomFilter/1.0/BloomFilter-1.0.jar it.unipi.
    hadoop.MapRedBloomFilter title.ratings.tsv output 157000 24404 63482
    171928 420900 990060 2117062 3578304 3406703 1092505 156103 7
    WithBloomFilters
```

`title.ratings.tsv` is the input file, `output` is the name of the output file, `157000` is the size of the partition to be handled to each mapper, the ten numbers that follow are the values for m for each filter, `7` is the value for k and `WithBloomFilters` is a flag that toggles which of the two implementations is to be run.

```
hadoop jar it/unipi/hadoop/BloomFilter/1.0/BloomFilter-1.0.jar it.unipi.
    hadoop.MapRedFalsePositiveRateTest title.ratings.tsv testOutput 157000
     output/part-r-00000 hadoop-namenode 9820
```

`title.ratings.tsv` is the input file, `testOutput` is the name of the output file, `157000` is the size of the partition to be handled to each mapper, `output/part-r-00000` is the file from which the saved Bloom filters have to be read, `hadoop-namenode` is the host and `9820` is the port.

The decision to set the partition size to 157000 was taken following the best-practice of assigning 1-1.5 cores to each mapper process; since the cluster that was used for testing had a total of 8 cores and the dataset had approximately 1.2M lines, 157000 was a suitable size for each partition. A more in-depth analysis on this parameter and its effects on the Key Performance Indexes is carried out in chapter 5.

Chapter 4

Spark

The second version was implemented in Python, using the Spark framework.
The directory structure of the project is the following:

```
hadoop
└── spark
    ├── BloomFilter.py
    ├── spark_compute_params.py
    ├── spark_bloomfilter.py
    ├── spark_FPR_test.py
    └── util.py
```

The files organization is quite similar to the one used in Hadoop, with the same classes used for the same purposes.
The hash function family is again MurmurHash, an implementation of which can be found in the `mmh3` Python module.
There was no need for a Python equivalent of the Java `NLineInputFormat` class: the `SparkContext` class available in the `pyspark` Python module already has partition size handling built-in within the `textFile()` method, which simply takes the number of partitions as additional argument when instantiating the first resilient distributed dataset (*RDD*).

Usage

The three `spark_*` files can be run with the following commands:

```
spark-submit spark_compute_params.py yarn hadoop-namenode 9820 title.
   ratings.tsv output 0.01
```

`yarn` is the *master* argument that selects on which FS the script has to be run, `hadoop-namenode` is the hostname, `9820` is the port, `title.ratings.tsv` is the name of the output file and `0.01` is the desired value for the FPR p.

```
spark-submit --archives pyspark_venv.tar.gz#environment spark_bloomfilter
    .py yarn hadoop-namenode 9820 title.ratings.tsv output 8 24404 63482
    171928 420900 990060 2117062 3578304 3406703 1092505 156103 7 0.01
    WithBloomFilters
```

`title.ratings.tsv` is the input file, `output` is the name of the output file, `8` is the number of partition the dataset has to be split into, the ten numbers that follow are the values for m for each filter, `7` is the value for k and `WithBloomFilters` is a flag that toggles which of the two implementations is to be run.

```
spark-submit --archives pyspark_venv.tar.gz#environment spark_FPR_test.py
     yarn hadoop-namenode 9820 title.ratings.tsv testOutput output/part
    -00000
```

`yarn` is the *master* argument that selects on which FS the script has to be run, `hadoop-namenode` is the hostname, `9820` is the port, `title.ratings.tsv` is the name of the output file, `testOutput` is the name of the output file, `output/part-r-00000` is the file from which the saved Bloom filters have to be read.

Chapter 5

Performance

This final chapter concerns the performance evaluation of the different implementations of the algorithms.
Deployment and testing were all performed on a cluster of four virtual machines provided by the University. Each of them was a Ubuntu machine with 2 cores and 8 GB of RAM.
The Key Performance Indexes (KPIs) that were studied were:

- the `Map output bytes`, i.e. the number of bytes output by the Mappers to be sent to the reducers. This impacts the amount of traffic to be sent over the network during the execution of the algorithm.
- the `wall time`, i.e. the total time elapsed between the launch of the program and the its completion, with the output written to HDFS

The tests were performed with different configurations, obtained by changing the following paremters:

- the FPR `p` (v. 2.1): its value ranges from 0.00001 (0.001 %) to 0.1 (10 %), incrementing each time by an order of magnitude
- the number of hashing functions `k`: its values range from 3 to 9, with increments of 2. Furthermore, configurations with no constraints on the value of `k` were tested.
- the number of mappers: its value ranges from 4 to 12 with increments of 2

Each test was run 10 times to limit the impact of outliers. This yielded a total of TODO tests.

5.1 Hadoop map output bytes

Map output bytes as function of p

Figure 5.1 shows the amount of bytes sent by the mappers to the reducers, as function of the FPR `p`, for both implementations and two different values of k.

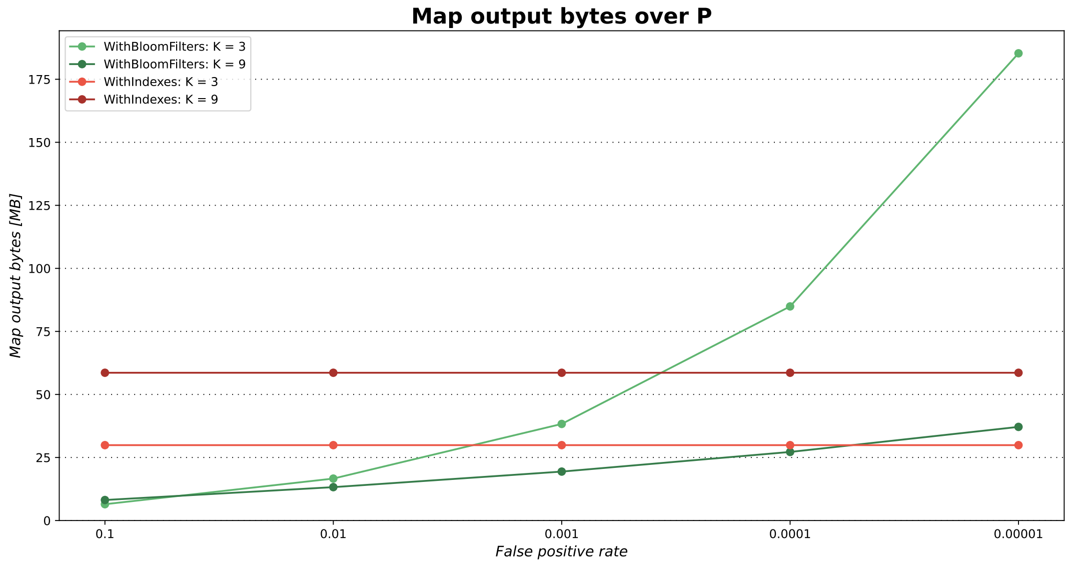

Figure 5.1: Map output bytes as function of `p`

As it stands out from the red lines, the amount of bytes produced by the mappers in the *withIndexes* implementation does not depend on p, but it only increases with `k`. This makes complete sense since, regardless of the size `m` of the Bloom filter, the number of indexes is always equal to `k`. Although the two values of `k` considered in the plot have a ratio of 3, the same ratio is not preserved in the values of the red line: this is probably due to some overhead bytes added by the serialization process of the objects and TODO.
On the other hand, in the *withBloomFilters* implementation, the amount of bytes sent by the mappers increases as `p` decreases, as it can be seen especially for lower values of `k` (light green plot in the figure). This is consistent with the first formula in 2.1. TODO

Map output bytes as function of p with optimal k

Figure 5.2 shows the amount of bytes sent by the mappers to the reducers, as function of the FPR `p`, for both implementations and an the optimal value for `k`.

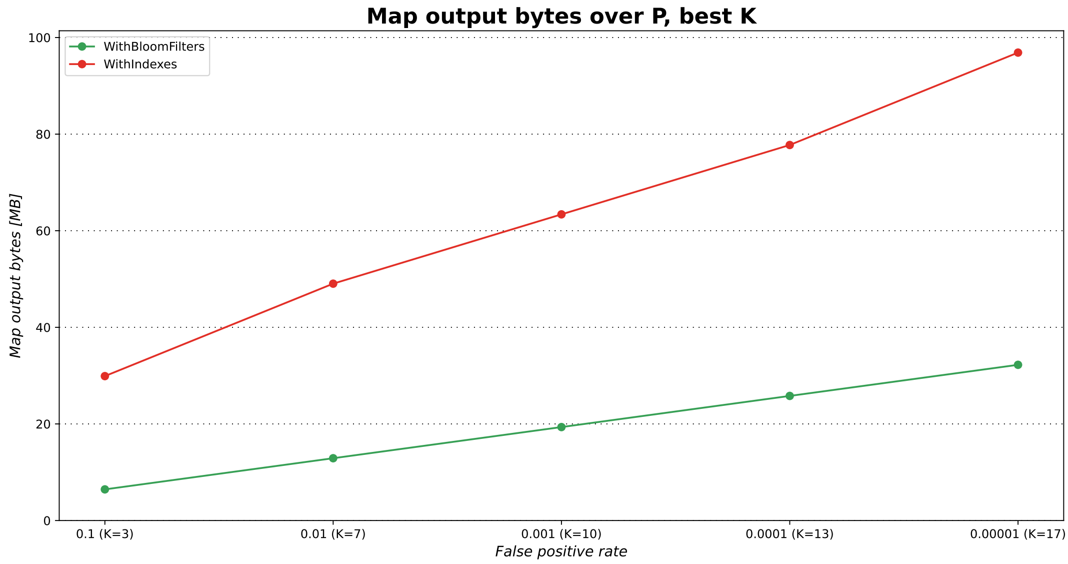

Figure 5.2: Map output bytes as function of `p` with optimal value of `k`

Both plots seem to follow a linear trend, which is consistent with the first and second formulas in 2.1: a geometric progression of `p` yields an arithmetic progression (because of the logarithm in the first expression) in the value of `m`. The optimal value of `k` is in turn linearly dependent on `m`.
These two relations explain, respectively, the linear increase in the size of bloom filters and the linear increase in the amount of indexes to be sent. CHECK

Map output bytes as function of the number of mappers

Figure 5.3 shows the amount of bytes sent by the mappers to the reducers, as function of the number of mappers with optimal values of `k`, for both implementations.

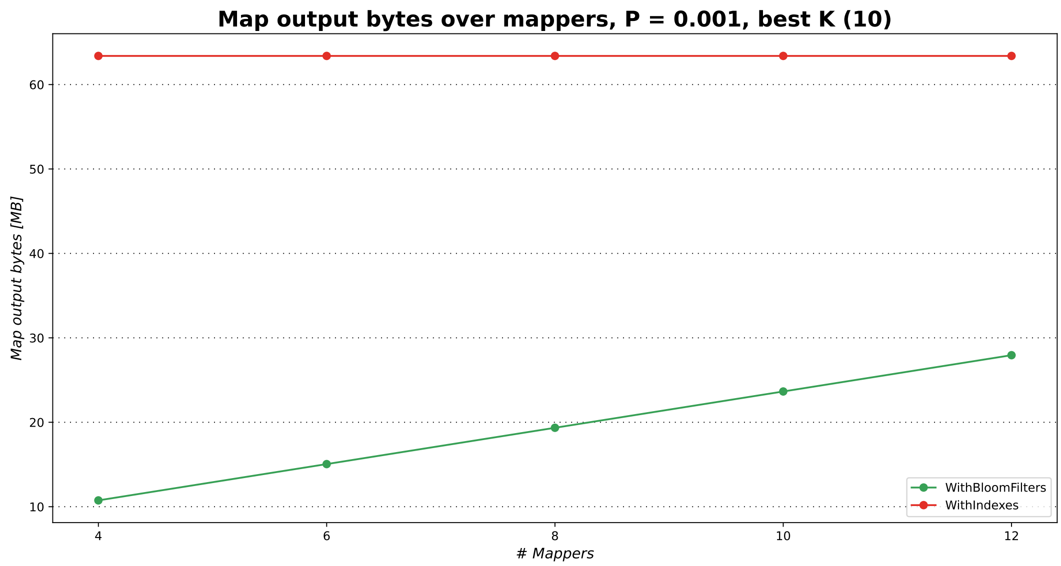

Figure 5.3: Map output bytes as function of the number of mappers

The explanation of this plot is quite trivial: in the *withBloomFilters* implementation every mapper has to create 10 Bloom filters and send them, regardless of how "full" they will be. Therefore, the overall amount of bytes sent by all the mappers to the reducer(s) will increase linearly with an increase in the number of mappers.
On the other hand, the *withIndexes* implementation always has a constant amount of traffic for fixes values of `p` and `k`, regardless of the number of mappers: this is because the total amount of indexes that are sent by all the mappers only depends on the size of the dataset. A bigger number of mappers will imply smaller partitions with fewer indexes being sent by each mapper; their overall amount will however be the same when they are received by the reducer(s).

5.2 Hadoop wall time

Wall time as function of p with optimal k

Figure 5.4 shows the amount take taken by the MapReduce algorithm to run from start to finish as function of `p`, with an optimal value for `k`, for both implementations.

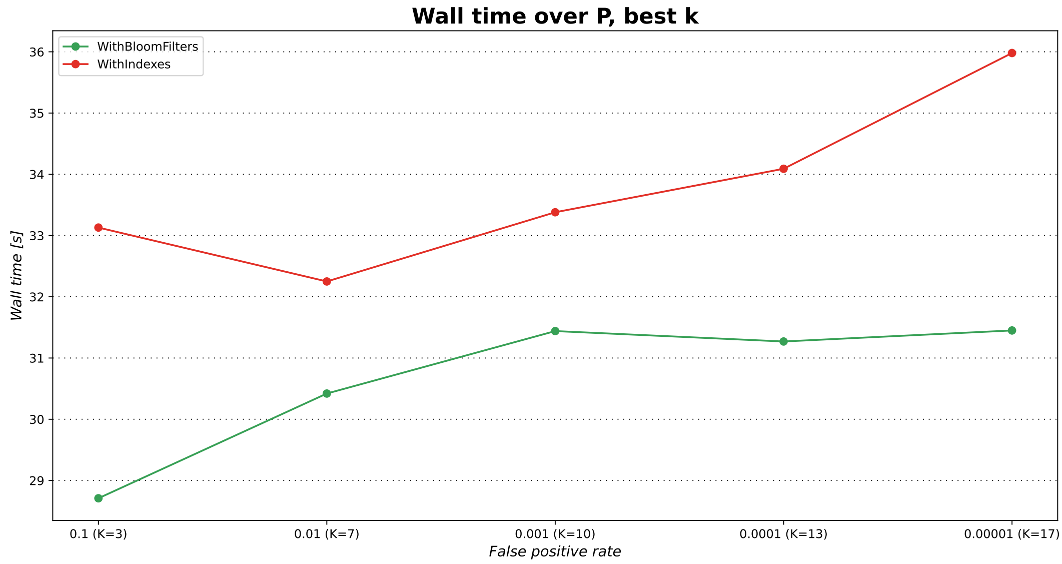

Figure 5.4: Wall time as function of `p` with optimal values for `k`

The plots are quite instable and noisy, which is likely caused by the fact that the cluster on which the tests were run was not a very stable and controlled environment and because of the low amount of replications of the tests.
Nevertheless it seems clear that the *withBloomFilters* implementation proves to be more efficient, time-wise, regardless of the value of the FPR.

Wall time as function of the number of mappers, fixed p and optimal k

Figure 5.5 shows the amount take taken by the MapReduce algorithm to run from start to finish as function of the number of mappers, with a fixed value of `p` and an optimal value for `k`, for both implementations.

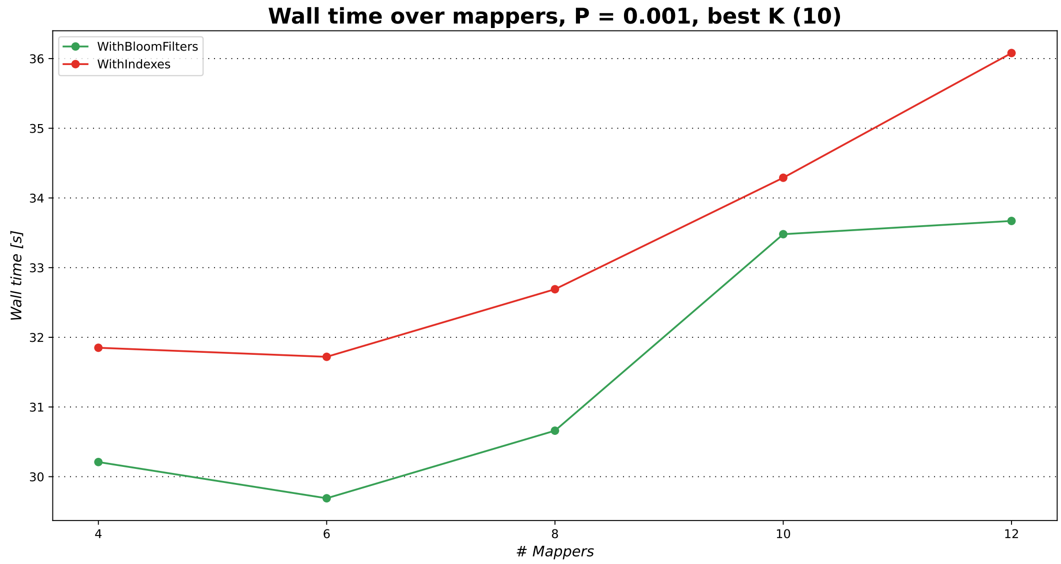

Figure 5.5: Wall time as function of the number of mappers

Both plots show a minimum for a number of mappers equal to 6. As mentioned in 3, the best practice suggests to assign from 1 to 1.5 cores to each mapper. The two minima in 6 correspond to 1.33 cores assigned to each mapper, which proves the best-practice to be well-founded.

5.3 Hadoop vs Spark

In this final section a brief comparison of the two implementations both in Hadoop and in Spark is presented.

Wall time as function of p with optimal k

Figure 5.6 shows the amount take taken by the MapReduce algorithm to run from start to finish, as function of `p`, with optimal values of `k`, for both implementations and both frameworks.

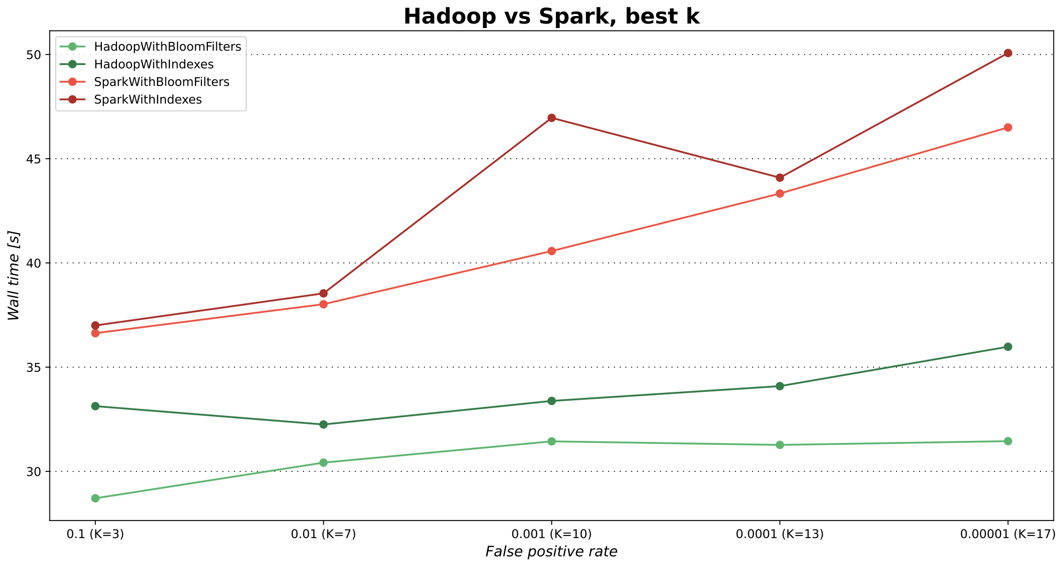

Figure 5.6: Wall time as function of `p` with optimal values for `k`, Hadoop vs Spark

Contrarily to what it was expected, Hadoop is faster than Spark on both implementations, despite using intermediate data writes on disks, as opposed to Spark which uses RAM.
This apparently abnormal behaviour was thought to be caused by the nature of the algorithm, which does not really take advantage of the efficiency of Spark on multi-passes. TODO aggiungere roba