CS-422 Project 2 Report

Francesco Intoci

EFPL, Spring Semester 2021

1 Introduction

This report aims to provide an insight on my choices for the implementation and design of the LHS algorithm used for Nearest Neighbors queries.

2 Exact NN

The ground truth for the system is represented by the **Exact NN** algorithm. This algorithm follows a naive approach, i.e for each provided query, it performs a cartesian product between queries and corpus RDDS and, for each resulting pair, it performs the Jaccard similarity between the lists of keywords. After that, those pairs with a Jaccard similarity below the threshold are discarded. Obviously, the algorithm is computationally expensive, since the complexity is $O(|queries| * |dataset|)$.

3 Base Construction

Base Construction exploits a LHS function, called **MinHash**, which takes as input an RDD of queries (made by a key-value pair, where the key is the film name and the value is a list of keywords describing the film), and outputs an RDD made by a key-value pair, where the key is the film title and the value is an integer, namely the signature of function MinHash. MinHash maps the list of keywords to an array of integers. Its signature is the *min* of such array. MinHash is applied both on queries RDD and data RDD. In particular, for data RDD, the MinHash signature is used in order to bucketize the films in the database based on their signature. Once the two RDDs are mapped into the *film name - MinHash signature* and *Set[Films]-MinHash signature* forms, an equi-join is computed between the two RDDs based on the equivalence of the signatures, i.e each film in the queries RDD will be joined with the corresponding bucket. This is a great improvement in terms of runtime on the ExactNN.

4 Balanced Construction

Balanced Construction tries to handle the skew problem which affects Base Construction: when the queries dataset is heavily skewed (i.e a MinHash signature is far more frequent than others), if using Base Construction, the Spark worker which will perform the join operation between the queries and the bucket with the skewed signature will become a struggler, leading to a drop in terms of performance. To solve this issue and maintain consistency in terms of runtime even in presence of skew, Balanced Construction introduces some overhead in order to try to split the workload fairly between workers.

The idea is the following: when receiving the queries dataset to evaluate, Balanced Construction will compute an histogram, where on the x-axis there will be the MinHash signatures found in the queries dataframe, and on the y-axis their frequencies. The histogram is defined as an array of integer pairs, matching each signature to its frequency. After that, based on a *partitions* parameter that is passed to the class constructor, the algorithm will compute the equi-depth of the histogram, i.e the number of queries that each partition should have:

$$ceil(\frac{\sum_i freq_i}{partitions})$$

After having computed the depth, the algorithm adopts a range partitioning approach to compute the bounds of the partitions: the histogram is ordered in ascending order based on the MinHash signature. By scanning linearly the histogram, a buffer accomulates the number of queries occured for that signature (the frequency). As soon as the buffer is greater or equal than the depth previously computed, the current MinHash signature is added to the bounds. The resulting bounds array will thus contain the **end** of the range of each partition. It is worth noticing that in literature, bounds often contains the start of the range, but I decided to implement it differently because it made more sense, for me, for the following part.

After computing the bounds, Balanced Construction will assign both queries and buckets, based on their MinHash signature, to the corresponding partition. This is done by simpling scanning the bounds array: as soon as a the MinHash signature is lower or equal to the current value of the bounds array, the assigned partition will be the index of the element in the bound array. It is worth noticing that, in the case the corpus RDD contains a MinHash signature which was not encountered during the scanning of the queries RDD, it will be added to the last partition.

The algorithm exploits a custom Partitioner class by calling the Spark partitionBy method on both the queries and buckets RDDs. The Partitioner will handle the mapping between MinHash signature and partition number. After that, the algorithm will perform a join again based on the MinHash signature.

4.1 Example of constructing bounds

Suppose that the histogram computation, on a given queries set, returns the following statistics:

$$histogram = [(id1, 1), (id2, 3), (id3, 1)]$$

Suppose that the balanced construction is created with a partitions parameter of 2. The resulting equi-depth of each partition is, accordingly, $ceil(5/2) = 3$.

Tha algorithm will thus scan the histogram (that is ordered), buffering the frequencies until the equi-depth is met. For example, when scanning id2, the buffer value will be $4 > 3$ and hence id2 will be selected as first bound. One the histogram is scanned, its final id is appended as final bound. The resulting bound for the example is: $[id2, id3]$, meaning the intervals $[id1; id2]$ and $(id2, id3]$. However, in the assignation step in the Partitioner, values above id3 that were not encountered in the queries dataset, will be assigned to last partition. The real range for last partition is thus $(id2, +inf)$.

5 BroadCast Construction

BroadCast Construction tries to further increase the runtime performance of Base Construction by reducing the shuffling operations needed. In particular, the construction is helpful when dealing with a large RDD of queries and a rather small corpus RDD. The algorithm is basically identical to the one used in Base Construction, but in this case the aim is to exploit the small dimension of the corpus and, hence, the buckets. The algorithm will collect the buckets RDD and, using a broadcast variable, broadcast it to all Spark workers. After that, the algorithm will simply call a map on the queries RDD, mapping each film to the set contained in the bucket with the same signature. It is worth noticing that my first implementation used a suboptimal data structure, i.e an array. Previously, each map operation on a tuple had to incur in a linear cost in the size of the buckets array, since I was performing a linear serach. My second implementation, rather than on a binary search, relies on a Map collection, created when calling the collect method on the buckets RDD with collectAsMap method. The increased cost when creating the map is shadowed by the gain when calling the map operation, since now the cost for retrieving the corresponding bucket is $O(1)$.

6 AND & OR Construction

When trying to increase the precision or recall of the Base Construction (which will inherently loss information when compared to the ExactNN approach), one can use respectively the **AND** and the **OR** constructions. These constructions, formed by a number N of simple or composite constructions (all with a differently seeded MinHash), the children, will iteretativelly call the eval methods of their children, stack their results, in form of RDDs, in one RDD, and then call

a reduceByKey method on the resulting RDD, using the film name as a key. The AND construction will perform the reduction by computing the intersection of the resulting sets for each key, while the OR will compute the union, in order to accordingly increase precision or recall. It is worth noticing that, in order to keep duplicates at file level (the provided queries.csv) and do not mix duplicates between results of children, before calling the eval method of the children, the composite constructions will append a random tag to the movie name, in order to identify tuples across resulting RDDs from children. The tag will be later removed when providing the result. Without this "trick", the reduceByKey method would have reduced also the duplicates that were contained in the original queries dataset.

7 RunTime Performance Evaluation

- As expected, ExactNN was the worst method in terms of runtime. Looking at the graph, it can be seen that the algorithm does not scale (hence it was pointless to test it with datasets bigger than the first one, given the already poor performance).

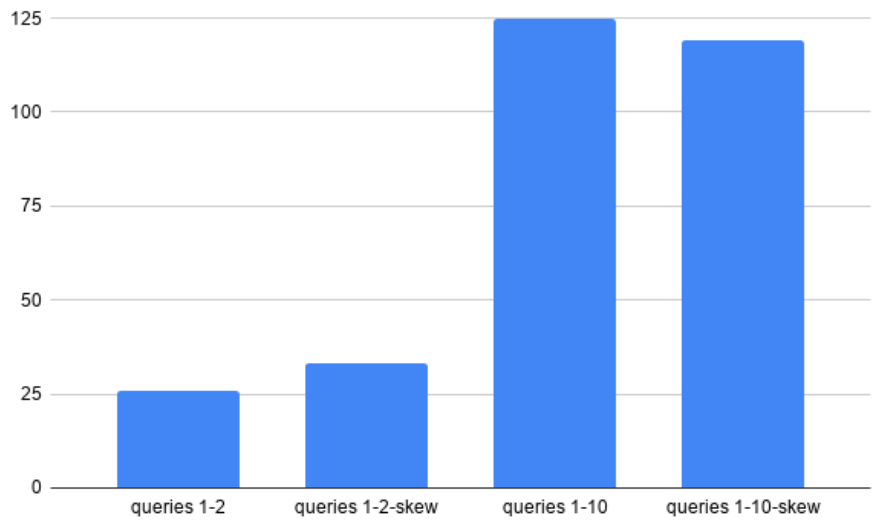

Figure 1: Exact NN runtime

- Base Construction showed a huge improvement in performance, when compared to ExactNN.

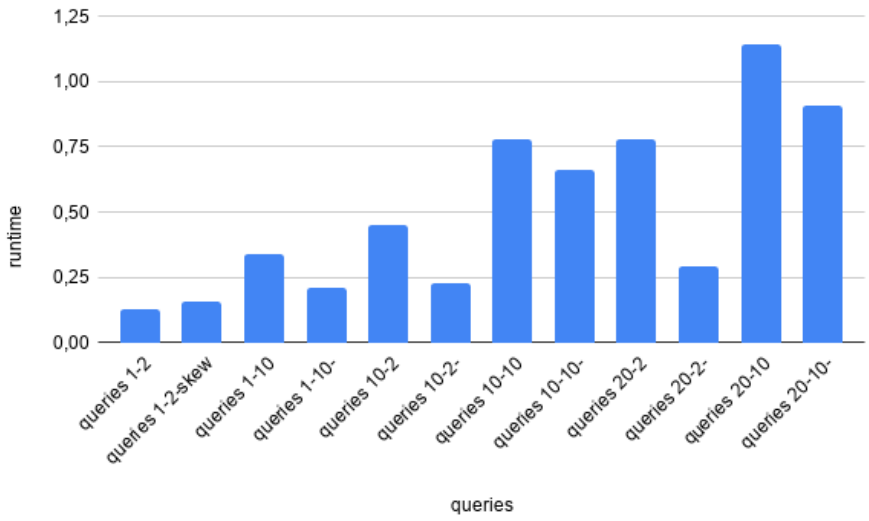

Figure 2: Base runtime

- Balanced Construction showed worse performance in respect to Base Construction. The probable explanation, also supported by the fact that, in general, skewed queries datasets did not led to a decrease in performance, is that the skew introduced by the datasets is not enough to put the Balanced Construction, taking into account the overhead for balancing the load, in a position of advantage in respect to Base Construction.

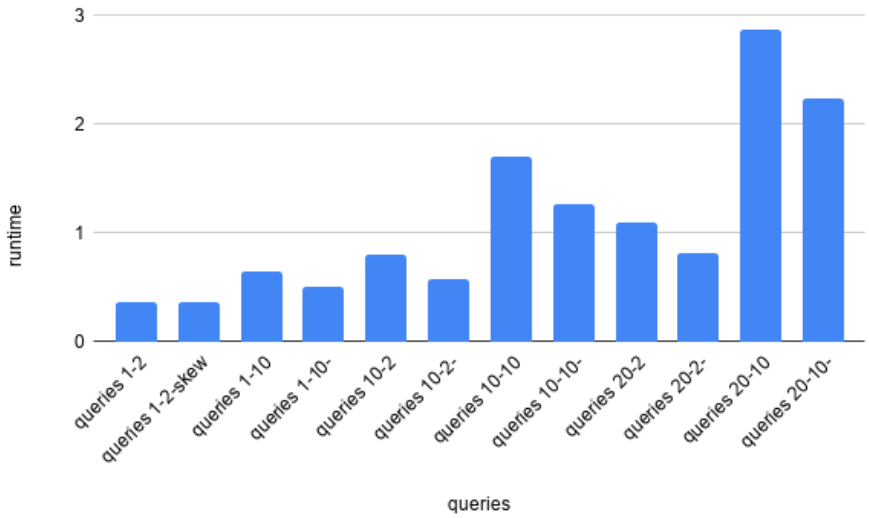

Figure 3: Balanced runtime

- Broadcast Construction proved to be better than the Base Construction in almost every test. Probably, the provided corpus datasets were not big enough to prevent the efficient broadcasting of the buckets.

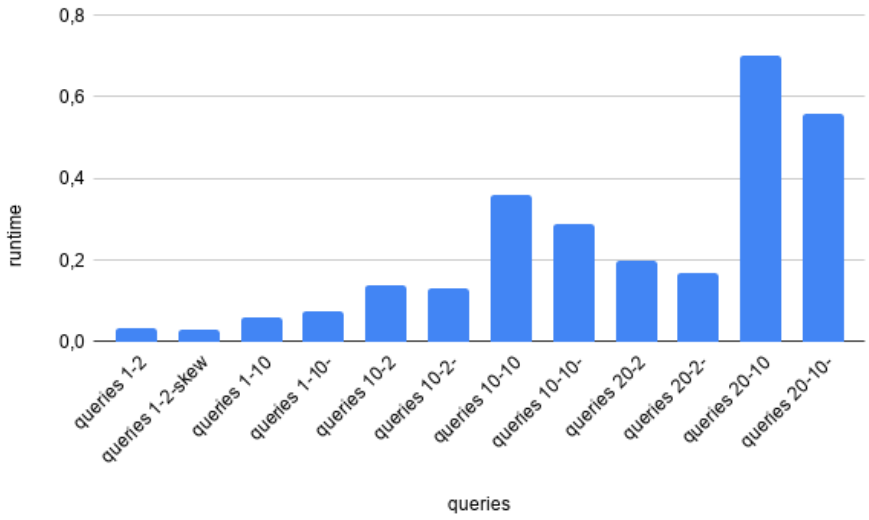

Figure 4: Broadcast runtime

8 Accuracy Evaluation

8.1 Disclaimer

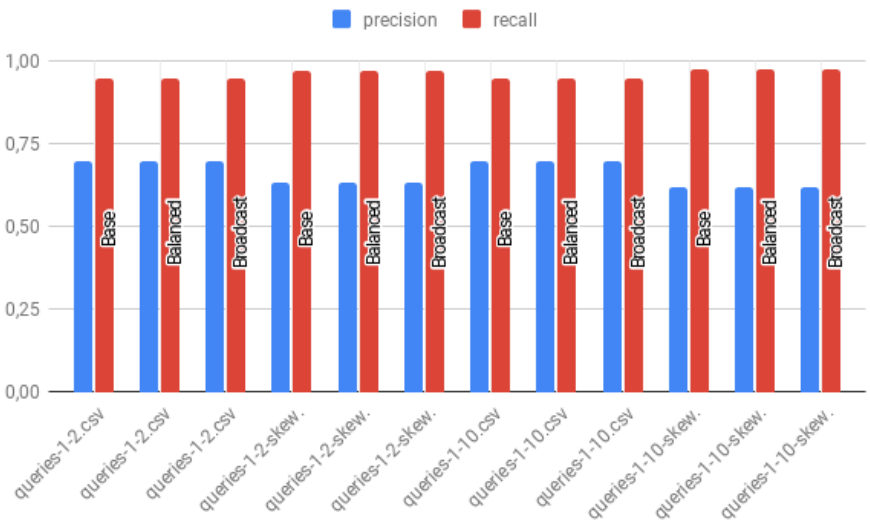

Figure 5: Comparison between base constructions

In the accuracy evaluation, I decided not to test the performance of Balanced and Broadcast BaseConstructions. The reason for that is, as it is shown in the following graph, their performance in terms of accuracy are the same. This is clearly an expected behaviour: the algorithms were designed to increase the performance of Base Construction in terms of runtime. dealing with *ad-hoc* scenarios like the presence of extreme skew in the queries or a large queries dataset querying a relatively small corpus. The underlying logic for finding the Nearest Neighbours (i.e the MinHash function), remains the same.

8.2 Average Distance

In the following graph, the average Jaccard Distance between each queried film, and the resulting Nearest Neighbours, is shown. The constructions that were compared are: ExactNN with a threshold for Jaccard Similarity of 0.55, Base Construction, AND Constuction with 6 childre, OR construction with 6 children. As expected, ExactNN was the most precise, followed by AND, Base and OR Constructions.

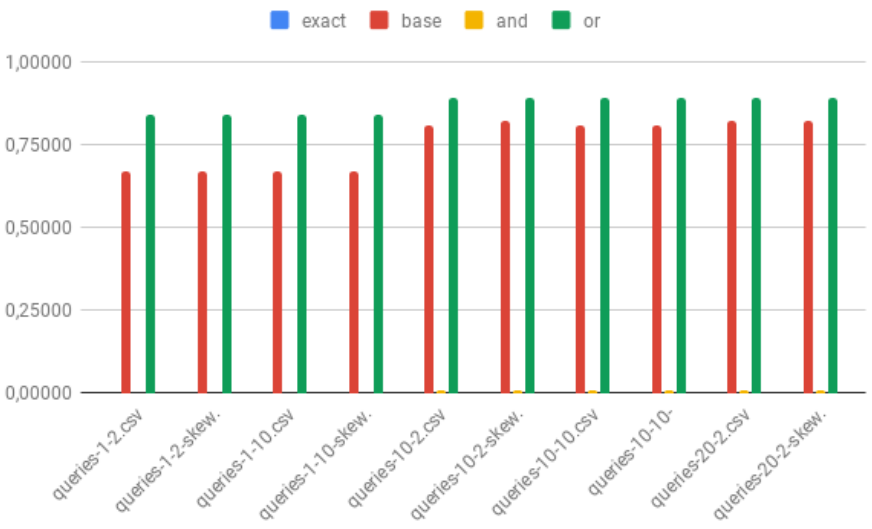

Figure 6: Average Distance, run on cluster. Value for ExactNN and AND are plotted, but the high difference in magnitude makes them not visible in the plot. For larger datasets, AND is visible in the bottom.

8.3 Accuracy and Recall

The following plot shows value for accuracy and recall. The tested constructions were again the four tested in Average Distance. As expected:

- Base Construction has poor performance in both precision and recall.
- OR Construction has very poor precision, while achieving the highest recall.
- AND Construction has low recall, while achieving the highest precision.

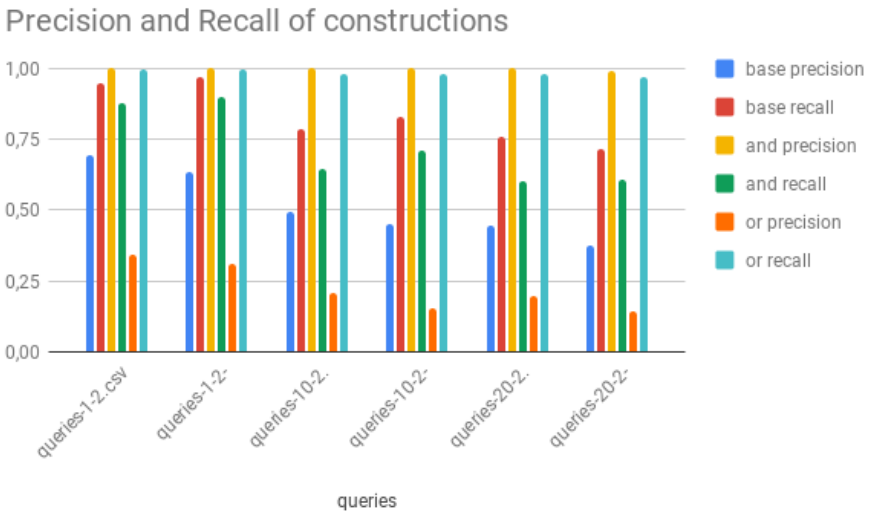

Figure 7: Precision and Recall test

9 Effect of Extreme Skew

When the extreme skewed dataset was provided, I decided to test whether the Balanced Construction was indeed performing better in case of extreme skew. As the results show, this was the case.

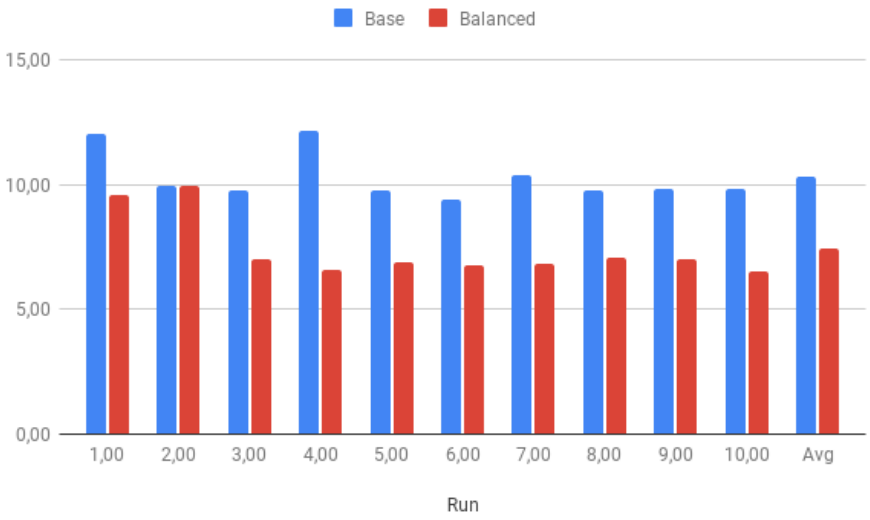

Figure 8: Base vs Balanced Runtime with skew