Introduction

Several optimisations have been considered and utilised in my implementation of the self-join set similarity algorithm. Two distinct chosen optimisations strategies were to transform variables to smaller sized datatype, such as transforming tokens into integers, and utilising filters to reduce the number of candidate pairs that required verification calculations.

Datatype Size Optimisation

Transforming the headlines into a set of integers provided several benefits. Firstly this reduced the memory used in Spark tasks. Since stop-words were removed from headlines, most tokens' string memory size occupied more memory than the 4-byte size of an integer. As such, using integers to store tokens could reduce the memory size of the Dataframes in the costly self-join and verification stages of the program.

Secondly, the integers assigned to words could be assigned based on their order of frequency within the document, saving the need to broadcast a token frequency list to nodes required during the prefix filtering stage. For prefix filtering to be performed, word tokens needed to be sorted in order of increasing frequency. To achieve this with tokens in string form would require a token frequency list to be broadcasted to each worker node. With tokens in integer form, the integer assigned to the word could be assigned based on its frequency in the document, avoiding the need for a broadcast variable that would occupy memory on worker nodes.

Beyond altering the types of tokens, all variable types were chosen based on the smallest sized type that that variable's range of possible values would fit into. For example, candidate pairs of headlines were required to be from different years, requiring information regarding years to be used in tasks. A reasonable assumption was made that years would not exceed 256 unique possible values, and as such the Byte variable type was used to store year data, rather than the integer type.

Filtering Technique Optimisations

Two filters were used to decrease the number of comparisons across the algorithm, namely a prefix and a length filter. By the use of a prefix filter, it could be ascertained that if a subset of sets r and s didn't contain any similar token between their prefix subsets, then those two sets could not have similarities above the given threshold. This is achieved by the fact that after more tokens are compared between two sets, the range of possible similarity values decreases. In the case of comparing two unsimilar sets against a known threshold, then there is a point at which enough tokens between the sets have been compared to confirm that the similarity cannot exceed the threshold.

For example, with a threshold of 0.8 and a set r where |r| = 10 and set s where |s| = 10, if we know that there exists three items in r that have no matching tokens in s, then at most r and s can have a similarity of 0.7. As such, there is no need to perform a formal similarity calculation between r and s, and the pair r and s is not considered a candidate pair. Furthermore, if uncommon items are examined first, there is a greater chance the between sets r and s three unsimilar tokens can be found. Sorting by frequency is used to achieve this, maximising the probability that sets do not become candidate pairs.

Along with prefix filtering, a length filter was utilised. In the case that two sets share a token between their respective prefix subsets, they are added to a candidate pairs table. However, if those two sets are not of similar enough length, it can also be ascertain that those two sets cannot pass the threshold. For example, with a threshold of 0.8 and a set r where |r| = 10 and set s where |s| = 7, even if r and s share a token in their prefix subsets, sets r and s cannot pass the threshold requirement, since at most they can have seven similar tokens giving a Jaccard Similarity of 0.7.

Additional Optimisations

Beyond utilising the aforementioned optimisations, general best practices were also used to decrease running time. For example, memory sizes of Dataframes were minimised before operations large join operations. During the prefix filtering task, a calculation of a prefix length of a set is required to prune the set into a subset. Crucially, the prefix length of a set r can be calculated from r alone as |r| - ceil(|r| * T) + 1. Because of this, sets can be transformed to prefix subsets before tokens are examined between sets, a task that requires a costly self-join operation.

Additionally, redundant operations were avoided during the algorithm runtime. For example, duplicate candidate keys were removed before similarity scores were calculated for candidate pairs, reducing overall runtime.

APIs were also used to optimise runtime. Excluding indexing tables and outputting the final result, the use of RDDs was avoided. In place Dataframes were used, taking advantage of the optimisations made by the Spark compiler during compile time. In the same vain Spark, SQL functions were used over User Defined Functions (UDF). Beyond this, native Spark functions have been substantially tested for optimisations and are likely faster than UDFs.