Big Data Assessed Exercise Report

Program logic summary

Our implementation successfully provides the top 10 documents based on the DPH score relevance ranking of a set of user-defined queries. It also takes into consideration duplicate results and returns only the top results with overly similar documents filtered out using a similarity score. It initially uses a broadcasted query list to broadcast the user defined queries for processing, using a filter function to filter null values on the **id, title, contents**, and **subtype** columns of news articles and initializes different accumulators (long, hashmap-based) for DPH score parameters.

The first spark transformation is a map function which takes in the query list and accumulators via its constructor and performs text processing (tokenization, stemming) on the news articles dataset. It also increments all passed accumulators for computing the query term frequency and total document size parameters. The filtered and processed news articles dataset in the form of a custom structure (*NewsArticleProcessed*) is then passed on to the next transformation along with the computed parameters which are stored in corresponding variables in the driver program. This transformation is a flatmap function which computes the DPH score for each query as an average of query term DPH scores and appends it into a custom structure list (*NewsArticleDPHScore*). To compute the query-wise DPH top 10 ranked documents, we first group the news article DPH structure by query key and use a flatmap groups transformation to get the output. The transformation sorts the documents in descending order of DPH scores, filters out the overly similar documents by their titles using a similarity score, fetches the top 10 DPH score ranked results per query and redefines it into the required *RankedResult* and *DocumentRanking* output structure.

Custom functions

NewsPreprocessor: This is a map function which is used to process and filter the Dataset of news articles wherein it calls the text pre-processor function to perform stemming and tokenization on the news articles and contents. It checks for presence of null values in the content and subtype and as a side process, it also computes the query term frequency in the entire corpus using accumulators, which is used to calculate the DPH score in the *NewsArticleDPHProcessor* function. It uses a custom defined Query Term Accumulator Map and a custom *NewsArticleProcessed* structure to represent and store the information related to the processed NewsArticle.

NewsArticleDPHProcessor: This is a flatmap function which computes the average DPH Score for a given query by using the processed news articles from the *NewsPreprocessor* function, document length, query term frequency and total corpus document accumulators. It computes the DPH score per query term on each document and averages the DPH scores for the entire query. It uses a custom structure *NewsArticleDPHScore* to represent and store the DPH scores by query along with other news article information and appends into a list to be returned as an iterator in the driver function.

QueryGroupsRanking: This is a flatmapgroups function which is used to group a *NewsArticleDPHScore* object by query and compute the top 10 DPH Score ranked documents per query. This function takes as input a *KeyValueGroupedDataset* with a Query object and *NewsArticleDPHScore* objects as key-value pair which it obtains from a groupbyKey function: *QueryKeyFunction*. This function performs sorting in descending order for the *NewsArticleDPHScore* by each article's DPH Score, calculating the similarity score between the titles of the articles and filters out the articles having high similarity score. During the iterations to compute similarity score, it converts the filtered articles along with their DPH scores into the required *RankedResult* structure and finally consolidates the top 10 ranked documents into the *DocumentRanking* structure by the query key. It returns a *DocumentRanking* list iterator which is returned to the driver function to obtain the final result.

Efficiency Discussion

We have used several approaches to improve the efficiency and scalability of the application. On a high-level, we have attempted to execute all required computational tasks within spark transformations for increasing parallelism and minimize the number of user-defined functions to reduce computational bottlenecks.

We have utilized broadcast variables for the Query object so that only a single copy is passed on to multiple functions - *QueryTermAccumulator*, *NewsPreprocessor* and *NewsArticleDPHProcessor*, hence reducing redundancy. We have also pre-computed the query term Frequency, total document length and total corpus documents in the *NewsPreprocessor* map function itself instead of defining separate reducer functions so that it can be directly used in the *NewsArticleDPH* Processor function for computing the DPH Scores. To execute this, we had to initialize a custom hashmap structure with the query term as key and an accumulator as its corresponding value. Instead of defining separate functions to convert our custom structures into the required output structures – *RankedResult* and *Document Ranking*, we iteratively create *RankedResult* objects and *DocumentRanking* object in the *QueryGroupsRanking* function so that it can be directly returned to the driver program.

We have avoided returning data and performing computations in the driver program wherever possible to minimize the memory usage. We have checked for the presence of null values in **id**, **title**, **contents** and **contents subtype** in the initial parts of the program itself in order to perform computations only on necessary data and improve efficiency. We have also avoided the use of for loops where possible to reduce computation time. We have used accumulators for the parameters of the function for computing DPH scores which eliminates the necessity of having to define separate local counters and functions and enables the global accumulator variables to pass data efficiently between transformations.

With the above optimizations and efficiency improvements, we were able to execute the entire program within 1 minute 3 seconds on the large news articles dataset.

Challenges

One major challenge that we faced was that the program was unable to run on the large dataset citing that the task result size was exceeded. We identified the root cause of the issue which was a *collectAsList* function fetching the entire filtered dataset into the driver program. We had implemented the *collectAsList* approach in response to another challenge, where we were unable to get the final value of the query term frequencies from an initial map function due to the next map function being run in parallel to the initial map function by the spark program. To resolve this, we had collected the resultant dataset from the initial map function into the driver program and fetched the last row to be passed on to the next map function, which unfortunately resulted in the task result size exceeded error. Eventually, we resolved this by initializing a hashmap based accumulator with query term as key and an accumulator to compute frequency of terms as value. This allowed us to eliminate the need of returning the dataset in the driver program and compute frequency terms efficiently.

This parallel computation of different map transformations was posing as a big impediment since we had dependencies between the functions. We were getting null pointer exceptions during the pre-processing of news articles and DPH score computations. We tracked this issue down to the fact that the latter transformations were executing the initial transformations again despite a spark action on the previous transformation. Accumulators (total documents) were getting computed twice for total documents due to the same. To resolve the null-values errors, we used the filter method which was both computationally fast and reduced the dataset size for more efficiency and lesser memory usage. To tackle the re-computation issue, we used optimized design to obtain only the relevant information in the driver program and passing it to the next map function.

We faced a challenge where we were unable to acquire the original *NewsArticle* object to define the final *RankingResult* and *DocumentRanking* objects. We tried to use join functions, but it was computationally expensive. As a resolution, we passed the *NewsArticle* object in the custom structures which we were passing through the map transformations, so we had the object while creating the resultant objects.

We encountered quite a few obstacles in the design considerations of transformation functions as well as the computation of parameters of DPH scores. We were very perplexed not only about the choice of transformation functions (map/flatmap/mapgroups etc.) and how to pass data through the transformations to compute the relevant entities, but also how to best optimize the code to have a minimum of transformations execute a maximum of tasks. We tried various different transformations such as reducers, mapgroups, flatmaps etc. until we arrived at a final set of transformations of map, flatmap and flatmapgroups to execute the entirety of the pipeline.

We also had a lot of challenges in using the Spark native Tuple2 data structure to store information as its implementation for storing the DPH parameters got increasingly complex, and hence we resolve this by choosing to use HashMap and accumulator combinations to execute our tasks.