Big Data: Systems, Programming and Management

Coursework

Jaskaran Singh Kawatra (2725957K)
Aditya Babu Kambalath Cherukovilakath (2767285K)
Hyun Suk Lee (2735512L)

Design Logic

To achieve the goal of getting the highest-ranked news articles for a given query using Spark's distributed architecture, we use several key transformations to ensure that our solution is reliable, scalable, and efficient. Our solution flattens the text in the content into a single paragraph using flatmaps, map the text to each document id, and tokenizes the text into individual words. The tokenized words are mapped to a TFinDoc object to enable efficient computation for term frequency within document and within corpus. The scores for each document query pair are then computed by transformations on the TFinDoc dataset to get the required parameters for each query. In this step, broadcasting is used to ensure that the datasets sent to each worker node in the cluster to calculate the parameters are read-only. After getting the scores for each document-query pair, we check for similar titles based on the provided textual distance function using a custom logic that uses hash sets to store seen titles. In the next step, the scores are mapped to the corresponding news articles using a flatmap. We then group the records using groupByKey where the key is the query string. Once the scores are grouped by query, the code computes the top matches for each query by mapping the scores to the query and document IDs, and then computing the match for each query based on the score. To attach the query to the final result, the code uses a broadcast variable that contains a map of the original queries and the associated metadata. This is broadcast to each worker node and used to attach the query to the final results in the required format.

List of Custom Spark Functions

1. NewsToContentMapper(): FlatMapFunction

This function is used to map each NewsArticle object to a list of Content objects, which contain the article's id, title, content, and subtype. it extracts the title and up to five paragraphs of content from each article, discarding other content such as images and captions. The function filters out any content that is not a paragraph and concatenates the text of the paragraphs into a list. The resulting content is returned as a single Content object with the subtype set to "paragraph", and the news ID and title are also included. The resulting content is used to build a corpus of text documents,

2. ContenttoString(): MapFunction

This function maps a Content object to a Tuple2<String, String>. This function is used to convert the contents of a Content object to a single String. The function first concatenates the paragraphs of the content into a single string with spaces in between, and then appends the title of the content to the beginning of this string. The resulting string is then returned as the second element of the tuple. The resulting dataset of Tuple2 objects is used to enable the mapping to generate tokens for each text document.

3. TextWithIdMapper(): MapFunction

This function maps a Tuple2<String, String> to a TextClass object. This function is used to convert the tuples produced by the ContenttoString function to TextClass objects, which are used to represent text documents in the dataset. The function simply takes the first element of the tuple as the ID of the text document, and the second element as the text content. These values are then used to create a new TextClass object. The resulting dataset of TextClass objects is used to generate tokens for each text document.

4. MakeTokens(): MapFunction

This function is used to convert an input object of type TextClass to an output object of type Tokens. In the call() method, an instance of the TextPreProcessor class is created, and the process() method is called on it to obtain a List of processed tokens from the input TextClass object. The ID field of the input object is assigned to the ID field of the output object. The processed_tokens list is assigned to the tokens field of the output object. This transformation takes a text string and creates a list of tokens that can be used to match against the query terms.

5. QueryDocumentFlatMap(): FlatMapFunction

The QueryDocumentFlatMap is a FlatMapFunction that takes in a Tokens object and returns an iterator of DocumentMatch objects. This function identifies the terms in the document that match the query terms, and creates a DocumentMatch object for each match. The QueryDocumentFlatMap constructor takes in two arguments: a HashSet of the query terms, and a Dataset of Query objects. The Query objects contain the original query and the terms in the query. These objects are used to create a mapping between the query terms and the original query.

In the call method, the Tokens object is used to extract the document ID and the list of tokens. For each token in the document, the function checks if it matches any of the query terms. If a match is found, a DocumentMatch object is created with the document ID, the matching token, and the original query. The function then returns an iterator of all the DocumentMatch objects that were created.

6. DocumentMatchtoTFinDocMapper(): MapFunction

A simple function that maps DocumentMatch objects to TFinDoc objects. The TFinDoc object is used in the final scoring step to calculate the relevance score for each document.

7. DocumentLengthMapper(): MapFunction

A function that maps a Tokens object to a DocumentLength object. It takes a Tokens object as input and calculates the number of tokens (i.e., the length) in the document, and then returns a DocumentLength object with the document ID and the length. This step is necessary to calculate the DPH scores.

8. DocumentLengthReducer(): ReduceFunction

This function is used to calculate the total length of a document by reducing the lengths of the individual tokens. It reduces a collection of integers to their sum.

9. DocScoreCalculator(): FlatMapFunction

This function calculates the DPH score for each document and term match. It takes several broadcast variables as input: totalDocsInCorpus, averageDocumentLengthInCorpus, termFrequencyInCorpus, and documentLengths.

The call method of the DocScoreCalculator class calculates the DPH score for each document and term match. It first retrieves the count of the matching term in the current document. It then retrieves the total term frequency of the matching term in the entire corpus. It also retrieves the length of the current document.

These values are then passed to the DPHScorer class, which calculates the DPH score for the document and term match. A DocScoresPerTerm object is returned, which contains the document ID, matching term, and DPH score.

10. TotalScoreCalculator(): MapGroupsFunction

This function calculates the total score of a document for a given query. The input to this function is an iterator over the DocScoresPerTerm instances for a single query-document pair. The function iterates through the scores, calculating the sum of scores and the number of terms that were matched in the document. Finally, it returns a Tuple3 containing the query-document pair, the total score, and the number of matched terms.

11. DocQueryScoreCalculator(): MapFunction

This function is used to calculate the score for each query-document pair. It takes in a Tuple3 of QueryDocID, Double, and Integer and outputs an AvgQueryScore object. It has a constructor that takes in a dataset of Query objects which is used to create a map of queries where the key is the original query text and the value is the Query object. It calculates the score as the total score divided by the number of terms in the query.

12. MapQueryDocIDKey(): MapFunction

This function transforms DocScoresPerTerm objects to QueryDocID objects by extracting the docid and query fields from the former and using them to create an instance of the latter. This function is used to create a new key for the DocScoresPerTerm objects that will be used to group them by document and query, allowing for the calculation of the total score and average score for each document-query pair. The resulting QueryDocID objects will be used as the key in the groupByKey operation that follows.

13. TitleExtractor(): FlatMapFunction

This function that extracts the titles of articles from the NewsArticle dataset. It takes a set of document IDs as input, and for each NewsArticle in the input dataset, it checks whether the document ID is present in the set. If the document ID is present, it extracts the title of the article and returns a tuple containing the document ID and title.

14. ArticleTitleAttacher(): MapPartitionsFunction

This function is used to attach the titles of the articles to the search results. It takes an iterator of AvgQueryScore objects and returns an iterator of AvgQueryScore objects. It uses a broadcast variable that contains a map of document IDs to their respective titles. It iterates through the input iterator, looks up the title of the document using the document ID from the input object, and sets the title on the object. Finally, it adds the object to a list and returns an iterator over the list. The output iterator contains AvgQueryScore objects with the document title attached.

15. RedundancyRemover(): Custom Function to Remove Redundancy

This function takes in a list of AvgQueryScore objects as input and returns a new list of AvgQueryScore objects with redundancy removed.

The method first creates a HashMap called queryToDocs that maps each query string to a list of AvgQueryScore objects that belong to that query. Then it initializes a list called results that will hold the final output and a set called seenTitles that will hold the titles of the documents that have already been added to the output list.

For each list of AvgQueryScore objects in the queryToDocs map, the method loops over the objects, checks if the score is greater than 0, and computes the minimum distance between the title of the object and the titles of the documents that have already been added to the output list. If the minimum distance is greater than or equal to 0.5, or if less than 10 documents have been added to the output list so far, the object is added to the output list and its title is added to the seenTitles set.

Once all the lists in the queryToDocs map have been processed, the method returns the results list.

16. AvgQueryScoreNewsArticleFlatMap(): FlatMapFunction

This function takes in a NewsArticle object and returns an iterator of ScoresWithArticle objects based on the relevant document ids present in the broadcasted avgQueryScoreMap. This function is used to enable grouping the DocumentRanking object based on the RankedResult object which contains an object of type NewsArticle as a field.

17. MapQueryKey(): MapFunction

This function is used to map the ScoresWithArticle objects to their respective query texts. It takes a ScoresWithArticle object as input, and returns a String representing the query text. The purpose of this function is to group the ScoresWithArticle objects by their query text in the groupByKey operation performed later in the code.

18. DocumentRankingCreator(): MapGroupsFunction

This function creates a DocumentRanking object for each group of ScoresWithArticle records with the same query. The call() method takes the query text and an iterator of ScoresWithArticle records and returns a DocumentRanking object. It first creates a Query object using the query text, which it gets from a broadcast variable containing a map of query texts to Query objects. It then iterates over the ScoresWithArticle records in the group and creates a RankedResult object for each one, using the docid, NewsArticle, and score. It then sorts the list of RankedResult objects by descending score. At the end it creates a DocumentRanking object using the Query object and the sorted list of RankedResult objects and returns it.

Efficiency Considerations

In this project, the data is partitioned in a way that takes into account the available memory on each worker node, so that the data can be loaded and processed more quickly. One of the key design decisions was to delay the search for the news article objects till the very end to ensure that the entire object is not loaded into memory. In the pipeline, we also make use of broadcast variables to efficiently share objects across the task of calculating the DPH scores for each document term pair. This is a key design decision as loading objects into memory for every computation of score for a document term pair would result in significantly reduced scalability of the system. We have also leveraged the use of efficient data structures like HashMap and HashSet to improve the performance of the pipeline. Unnecessary data is filtered out as early as possible in the pipeline to reduce the amount of data that needs to be processed. For example, only the subset of articles that match the given query is processed, and only relevant query terms are used to calculate scores. Certain intermediate results, such as tokenized documents and term frequencies, are also cached in memory to reduce the amount of disk I/O and computation required when these results are needed again in subsequent stages

Challenges Faced:

One of the significant challenges we faced was that our initial implementation to search for document ids with relevant scores was loading the entire NewsArticle object into a HashMap. Even though we tried to make it efficient by broadcasting the dataset so that the document match can occur based on document ids in the query, we were still unsuccessful in our approach as we had an out-of-memory error when we tried this approach. After further investigation, we found that setting up the HashMap in the first place can be an expensive operation on large datasets. The cost of creating the HashMap outweighed the benefits of transferring it as a broadcast variable. We then changed our approach such that the NewsArticle object was converted into a ScoresWithArticle dataset using a FlatMapFunction. This allowed us to iteratively compare the document ids from the broadcasted avgQueryScore map without having to load the entire NewsArticle object into memory.