Parallelized Okapi BM25 in Haskell

COMS4995 002 Final Report

Han-Ju Tsai (ht2572)
Ian Pan (ip2399)

1 Abstract

This paper documents the parallel functional programming implementation of Okapi BM25 in Haskell. We showcase both the accuracy of our program with respect to the high-ranking results returned, as well as performance improvements when the program is run under multiple cores.

2 Introduction

We are interested in the mechanics of ranking functions used by search engines that estimate documents' relevance for a given query. In natural language processing and information retrieval, we have learned about tf-idf, which is a popular weighting scheme for ranking and scoring documents used by search engines. With inspiration taken from a similar effort made by two students in Professor Edward's 2020 class, who parallelized the search ranking function tf-idf (Ye et al., 2020), we explore the intricacies of an even stronger ranking function, Okapi BM25, and apply it to a parallel setting. In the following sections, we give a brief overview of the search ranking function Okapi BM25, followed by our parallelized implementation, detailed by the chunking methods and results on multi-core tests. We also compare our program results with a benchmark on various examples to demonstrate the correctness of our implementation.

3 Okapi BM25: A Search Engine Ranking Function

In Okapi BM25, we can roughly divide its work into two stages. The first (preprocessing) stage consists

of building indices for the documents at hand, in which we record the occurrences of words both inter-document and intra-document. The second stage copes with scoring each document with respect to an input query. We apply the ranking function to each document, and output the top matched documents as our result.
We will count the number of occurrences of each word in the documents (term frequency) and how many documents include that word (document frequency). With the term frequency and document frequency calculated, the BM25 score of a document D can be determined..

The following formula produces the score of a document D for a given query Q.

$$\text{score}(D,Q) = \sum_{i=1}^{n} \text{IDF}(q_i) \cdot \frac{f(q_i,D) \cdot (k_1+1)}{f(q_i,D) + k_1 \cdot \left(1 - b + b \cdot \frac{|D|}{\text{avgdl}}\right)}$$

, where f is the term frequency of the word in document D. In addition, k_1 and b are empirically chosen hyperparameters.

In the formula above, note that IDF stands for “inverse document frequency”. IDF diminishes the weight of terms that occur very frequently in the document set and increases the weight of terms that occur rarely. For instance, IDF can lessen the importance of the stop-words such as “the”, “this”, and “that”. Without such offset from inverse document frequency, term frequency might over-emphasize the importance of these insignificant stop words and disrupt the balance of the score calculations.

4 Comprehensive Walkthrough of the program

4.1 Preprocessing the dataset

All the experimentations run on our program are done on a public stories archive (Archive Textfiles, n.d.). The dataset comprises 424 documents, including contemporary poems, Aesop’s Fables, and several collections of Sherlock Holmes’ adventures by Sir Arthur Conan Doyle.

In the preprocessing phase, each sentence is split into words, and each word is represented as a token. To ensure correctness and robustness, we cleaned up the dataset by converting characters to lowercase in each story and removing all irrelevant punctuations. This allows the later tokenization (splitting by whitespace) to be done efficiently, without having to worry that words

followed by a comma or period would be treated as a different token. Similarly, we also tokenize and normalize the input query similarly, as follows:

```
-- Return a list of normalized tokens
tokenizeAndNormalize :: String -> [String]
tokenizeAndNormalize query = map sanitizeWord (words query)
 where sanitizeWord word = map toLower (filter isLetter word)
```

4.2 Building Individual Term Frequency Hashmaps

We build a term frequency hashmap (i.e. tfMap) for each document. Each hashmap consists of the number of occurrences that a word appears in a specific document and is stored at the document level as a field of the Document data structure. The size (i.e. number of unique keys) of the hashmap would be the number of the unique vocabularies in the document. We can subsequently derive the variable termSet from the keys of the hashmap, once its construction is complete. The implementation can be shown as follows:

```
readDocument :: String -> String -> Document
readDocument title' text' = Document title'
                                     text'
                                     tfMap
                                     (Map.keysSet tfMap)
                                     (sum (Map.elems tfMap))
 where tfMap = buildTFMap text'

buildTFMap :: String -> Map.Map String Int
buildTFMap docText =
 Map.fromListWith (+) [ (word, 1) | word <- tokenizeAndNormalize docText ]
```

4.3 Obtaining A Global Document Frequency Hashmap

After we preprocess every document to a cleaner format, we build a global document frequency hashmap for the program, where keys are each unique word across the entire archive, and the values are the number of documents that contain the word. Unlike the term frequency hashmap which is calculated with respect to each document, and stored at the document level, the document frequency hashmap (i.e. dfMap) is a global variable that stores each word that occurs in the entire corpus along with their document frequency, i.e. the number of documents that the word appears in.

The programmatic implementation is shown as follows:

```
-- Get the document count (DF) for each word
getAllDF :: [Document] -> Map.Map String Int
getAllDF docs = Map.fromListWith
 (+)
 [ tup | doc <- docs, tup <- zip (Set.toList (termSet doc)) [1 :: Int, 1 ..] ]
```

4.4 Obtaining the top K stories

To obtain the top K stories, we build a min heap with size K. We loop over all documents and push a document into the min Heap at a time. As soon as the size of min heap exceeds K, we immediately pop the minimum one out from the heap. The size of the heap can be guaranteed to be at most K, so the operation of popping an element from the heap takes O($log\ K$) and results in the overall time complexity to be O($N\ log\ K$). Since the K is empirically much smaller than N, our algorithm will perform significantly better than naive sorting.

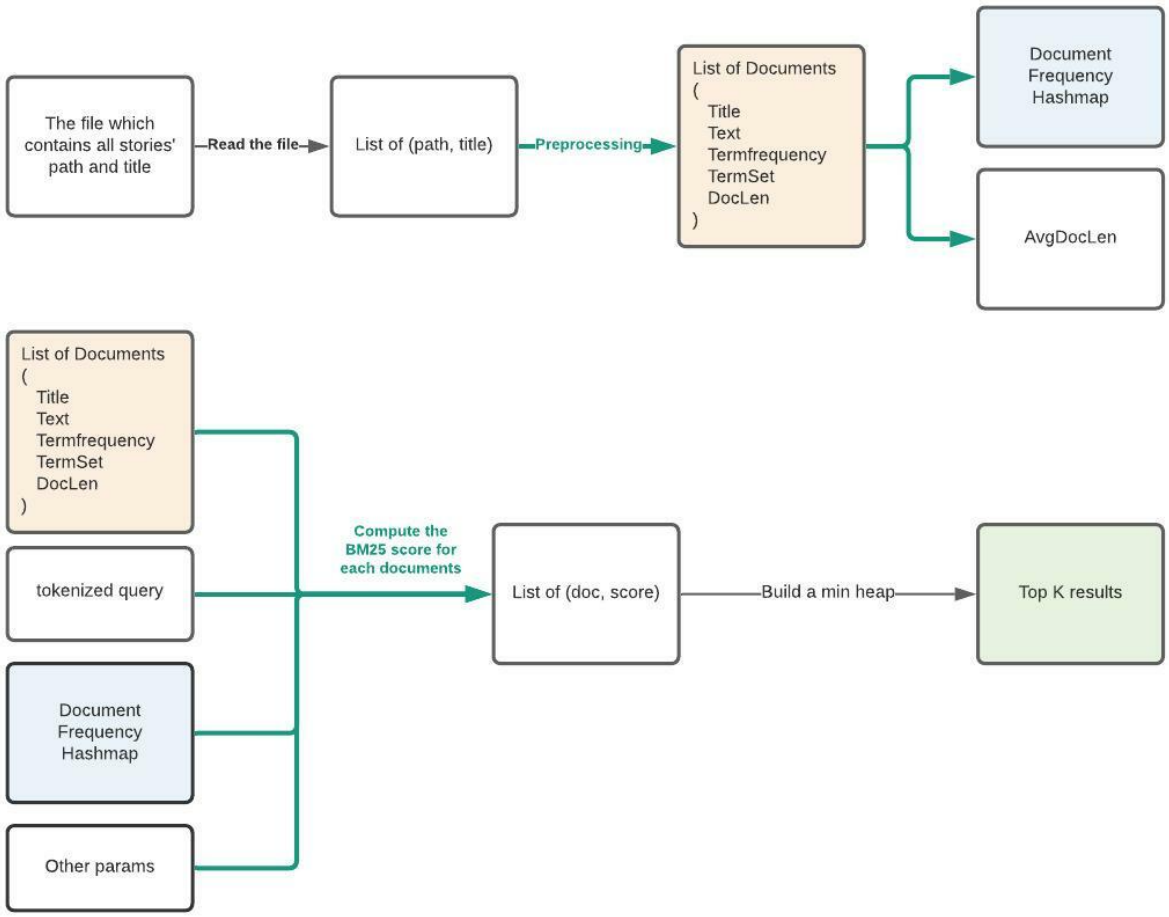
```
topKDocuments
 :: Int -> [(Double, Document)] -> Heap.MinPrioHeap Double Document
topKDocuments _ [] = Heap.fromList [] :: Heap.MinPrioHeap Double Document
topKDocuments topK (sd : sds) = maybePop
 (Heap.insert sd (topKDocuments topK sds)) where
 maybePop
   :: Heap.MinPrioHeap Double Document -> Heap.MinPrioHeap Double Document
 maybePop minHeap | Heap.size minHeap > topK = Heap.drop 1 minHeap
                  | otherwise                = minHeap
```

5 Parallelization

The following graph shows the architecture of our algorithm. The thick green arrows represent the implementation of parallelization. There are two flow/stages in the graph, which represent the data pipeline to process the BM25 model. Initially, we have a single text file containing each archived story's relative path and their title, separated by a comma with one such instance on each line. We process the text file as a list of "(path, title)" pairs and further read each separate file into a data structure named Document. The Document data structure contains five fields,

namely the document title, the entire text string of the document, a term frequency hashmap, an unordered set of all vocabularies of the document, and the document length. Once we have a list of such documents, we can compute various utility global variables such as the Document Frequency Hashmap and the average document length.

The second stage demonstrates how our search engine can be used. The program takes in a tokenized query as the input. With the pre-calculated model parameters, we use the aforementioned formula to compute the score. In the end, we loop over the list and maintain a heap with size k. Each time we will push a (document, score) pair into the heap. Once the heap size exceeds K, we pop the top out from the heap. The construction and balance of the heap have a time complexity of $O(N \log K)$, where N is the total number of documents, and K is the number of top results we are interested in.



5.1 Parallelizing with rseq and rpar

The runOkapiBM25 function in our main.hs file takes advantage of parallelization when calculating the average document length and obtaining the global document frequency hashmap after each document is read and parsed into Document objects. The following code snippet shows the parallelization implemented with rseq and rpar:

```
let (avgDocLen, dfMap) = runEval $ do
      avgDocLen' <-
        rpar $ (sum [ docLen doc | doc <- documents ]) `div` corpusSize
      dfMap' <- rpar $ getAllDF documents
      _      <- rseq avgDocLen'
      _      <- rseq dfMap'
      return (avgDocLen', dfMap')
```

5.2 Chunking with parListChunks

We adopt the chunking strategy to ensure that each CPU has an even workload. To be specific, we use the built-in parListChunks that allows us to sequentially apply a strategy to chunks of a list in parallel. The following code splits a Haskell list into chunks of length `chunks`, and processes the list using the function `getDocWithScore`. To balance the load, threads fetch work from a work queue (a sparkpool) (Stewart, 2016). Hence, the allocation to each thread is dynamically determined at runtime.

```
scoreAllDocs
 :: Int
 -> String
 -> [Document]
 -> Map.Map String Int
 -> Int
 -> Int
 -> Double
 -> Double
 -> [(Double, Document)]
scoreAllDocs chunks query docs dfMap corpusSize avgDocLen k b =
 map
     (\doc -> Data.Tuple.swap
       (getDocWithScore (tokenizeAndNormalize query)
                        doc
                        dfMap
                        corpusSize
```

```
                    avgDocLen
                    k
                    b
      )
    )
    docs
  `using` parListChunk chunks rseq
```

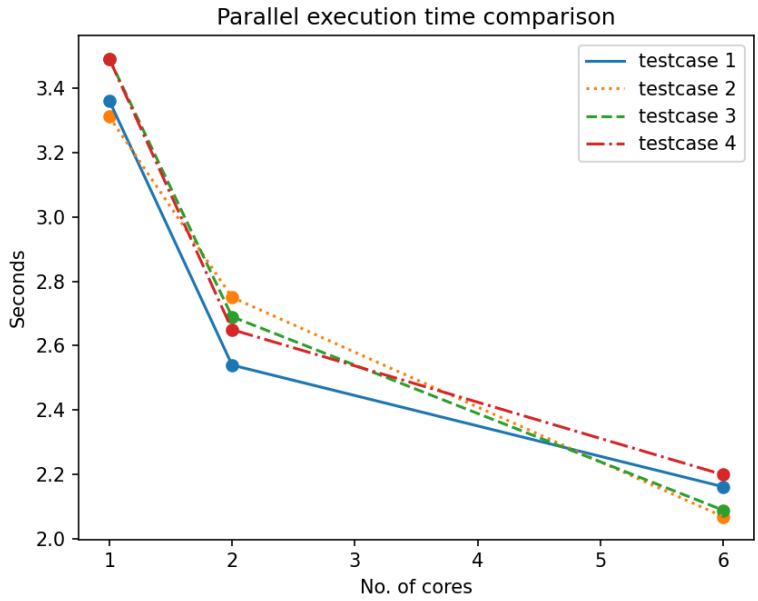
5.3 Multi-core Tests

To evaluate the effectiveness of parallelism, we perform experiments on 1, 2, and 6 cores, with four different queries. The four queries are: "*Sherlock Holmes*", "*Not quite so lazy, the second little pig went in search of planks of seasoned wood*", "*Outside, Weaver asked. The rain had almost stopped, but we stayed under the awning in case the acid content in the remaining drizzle was dangerously high. The closed door behind us cut off Pete's threats and curses.*", and "*An enemy at its gate is less formidable, for he is known and carries his banners openly.*".

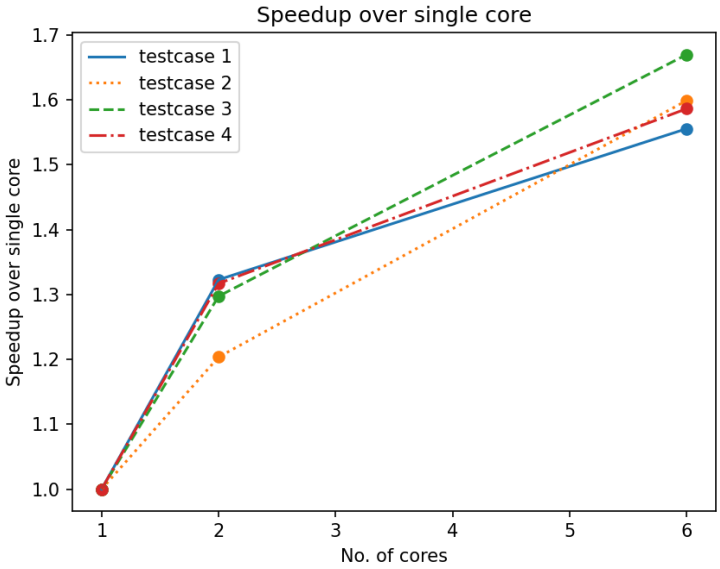
The table and the graphs below showcase the decrease in runtime while more cores are used to run the program. This phenomenon holds true for all 4 example queries that we input to the program. This proves the improvement in efficiency of our parallelized code.

Test Case \ No. Core	1	2	6
Sherlock Holmes	3.36	2.54	2.167
Not quite so lazy, the second little pig went in search of planks of seasoned wood	3.31	2.75	2.07
Outside, Weaver asked. The rain had almost stopped, but we stayed under the awning in case the acid content in the remaining drizzle was dangerously high. The closed door	3.49	2.69	2.09

behind us cut off Pete's threats and curses.			
An enemy at its gate is less formidable, for he is known and carries his banners openly.	3.49	2.65	2.20



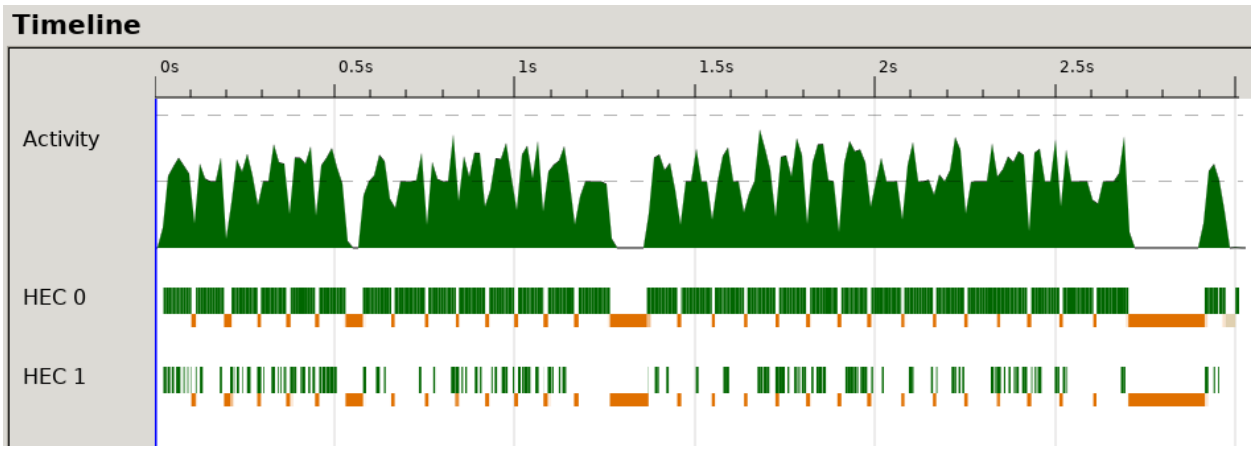
With respect to the speedup over single cores, we take the inverse of the runtime of multi-cores by single-core-runtime, and calculate the fraction of improvement as follows:



5.4 Threadscope Event Log

In the following event log, we use two cores as an example to showcase the parallelism of our algorithm implementation. When running the code on six cores, we notice that despite our parallel implementation, the program does not take full advantage of all cores. We suspect that this is because 424 documents are not a sufficient size of dataset to push the program to the limit. In other words, the Okapi scores can be efficiently calculated with two or three cores. This is also acknowledged in the final section "Conclusions & Future Steps" as a potential improvement to be made in the long run.

```
SPARKS: 30 (28 converted, 0 overflowed, 0 dud, 0 GC'd, 2 fizzled)
```



6 Comparison with Benchmark: rank_bm25

In this section, we compare our search ranking implementation with the Python package rank_bm25 (Brown, 2019) as our benchmark to prove the correctness of our program. One may notice slight differences between the results, specifically in order. This is due to the differences in handling negative IDF scores, and the hyperparameters k and b in the Okapi BM25 formula.

In rank_bm25, the package author calculates the inverse document frequency for a query term q_i with the following formula:

$$IDF(q_i) = ln(\frac{N - n(q_i) + 0.5}{n(q_i + 0.5)})$$

This results in a high chance of getting a negative value for the IDF score. The package author handles it by changing every negative IDF score to the average inverse document frequency multiplied by a hyperparameter "epsilon". This method inevitably loses information, since we will no longer be able to compare documents with various negative values – they will all end up with the same adjusted average.

In our Haskell implementation, we handled negative IDF scores more robustly by adopting the following formula:

$$IDF(q_i) = ln(\frac{N - n(q_i) + 0.5}{n(q_i + 0.5)} + 1)$$

This results in mostly positive IDF scores, and we can still rank the documents whose inverse document frequency would have been negative in rank_bm25. This preserves the relativity of suboptimal documents and maintains the robustness of the algorithm.

Our first example consists of the search query "Sherlock Holmes". The top 10 story titles returned by our parallel Haskell program are as follows:

"The Holmes List: A List of Books That Reference or Discuss Sherlock Holmes",
"The Adventure of the Bruce-Partington Plans",
"The Adventure of the Six Napoleons",
"The Story of the Five Orange Pips",
"The Adventure of the Missing Three-Quarter (A story)",

"The Hound of the Baskervilles",
"The Adventure of the Empty House",
"The Adventure of the Three Gables",
"The Adventure of Black Peter",
"The Adventure of the Golden Pince-Nez"

On the other hand, the top 10 story titles returned by the rank_bm25 benchmark is as follows:

"The Holmes List: A List of Books That Reference or Discuss Sherlock Holmes"
"The Hound of the Baskervilles, by Sir Arthur Conan Doyle"
"The Adventure of the Bruce-Partington Plans"
"The Adventure of the Six Napoleons"
"The Story of the Five Orange Pips"
"The Adventure of the Missing Three-Quarter (A story)"
"The Adventure of the Empty House"
"The Adventure of Wisteria Lodge"
"The Adventure of the Golden Pince-Nez"
"The Adventure of Black Peter"

In our second example, we experimented with a longer search query: "Not quite so lazy, the second little pig went in search of planks of seasoned wood". The top 10 story titles returned by our parallel Haskell program are as follows:

"The Story of the 3 Little Pigs"
"Aesop's Fables Translated by George Fyler Townsend"
"The Seven Voyages of Sinbad the Sailor"
"Interview with the Radar Ranger: A Work of Fiction by D. Railleur"
"Aesop's Fables (84 of Them) from The PaperLess Readers Club"
"Gulliver's Travels, by Johnathan Swift (1726)"
"The Heart of Darkness by Joseph Conrad"
"The Vigilante"
"Complete Transcription of Douglas Adams' Life, the Universe and Everything"
"Complete Transcription of Douglas Adams' Restaurant at the End of the Universe"

Similarly, the top 10 story titles returned by the rank_bm25 benchmark is as follows:

"The Story of the 3 Little Pigs",

"The Seven Voyages of Sinbad the Sailor",
"Aesop's Fables (84 of Them) from The PaperLess Readers Club",
"Aesop's Fables Translated by George Fyler Townsend",
"The Diary of the Swarves",
"The Story of Pinocchio",
"Once Upon a Mattress",
"Book 'Em Issue #3 (1995)",
"The Musgrave Ritual (Story)",
"A List of Father Good Stories"

7 Conclusion & Further Steps

This report focuses on the parallel implementation of Okapi BM25 in Haskell. The first two sections introduce our motivation and background of choosing this topic as our project. In section 3, we give a brief overview of the Okapi BM25 formula, as well as the intuition between choices of components in the algorithm. Section 4 provides a walkthrough of the algorithm implementation, starting from normalizing and tokenizing the dataset, to constructing term frequency and document frequency hashmaps, to finally using a min heap to obtain the top results as our program output.
In Section 5, we dive deeper into the parallelization implementation of our program. We start by explaining the motivation behind using rseq and rpar, and later move on to discuss chunking with parListChunks. In addition, we demonstrate the improvements in runtime in section "Multi-core Tests" with two plots showing the decrease in runtime when we increase the number of cores for our program. Finally, we show the threadscope event log to highlight the parallelization on an instance of two cores.
In Section 6, we compare our Haskell program with the public Python package rank_bm25, to ensure the correctness of our document rankings when run on a similar dataset. Despite the slight differences in implementation, we exhibit the correctness of our code both quantitatively and qualitatively through four example queries of different lengths.

Future works include pushing the limit of our implementation and running the program on a much larger dataset to see more drastic improvements of the parallelization. We plan to empirically quadruple the total runtime of a query to roughly 10 seconds.

8 References

Archives Textfiles. (n.d.) Retrieved from http://archives.textfiles.com/stories.zip

Brown D. (2019) "rank_bm25". Retrieved from https://github.com/dorianbrown/rank_bm25

Stewart, R. (2016) "Data parallelism with Haskell strategies and OpenMP". Retrieved from https://www.macs.hw.ac.uk/~rs46/posts/2016-05-29-skeletons-parallel-pragmas.html

Ye S. & Scanteianu D. (2020) "Parallelizing A Search Engine". Retrieved from http://www.cs.columbia.edu/~sedwards/classes/2020/4995-fall/reports/TFIDF.pdf

Appendix I: Program Listing

OKapiBM25_util.hs

```haskell
{-# LANGUAGE DeriveGeneric #-}
module OkapiBM25_utils where
import           Control.Parallel.Strategies
import           Data.Char
import qualified Data.Heap                    as Heap
import qualified Data.Map.Strict              as Map
import           Data.Maybe
import qualified Data.Set                     as Set
import           Data.Tuple
import           GHC.Generics                  ( Generic )


-- termFrequency: frequency of a word in this document
-- termSet: all the words in this Document (i.e. all keys of the termFrequency map)
data Document = Document
 { title         :: String
 , text          :: String
 , termFrequency :: Map.Map String Int
 , termSet       :: Set.Set String
 , docLen        :: Int
 }
 deriving Generic

scoreAllDocs
 :: Int
 -> String
 -> [Document]
 -> Map.Map String Int
 -> Int
 -> Int
 -> Double
 -> Double
```

```haskell
 -> [(Double, Document)]
scoreAllDocs chunks query docs dfMap corpusSize avgDocLen k b =
 map
     (\doc -> Data.Tuple.swap
       (getDocWithScore (tokenizeAndNormalize query)
                        doc
                        dfMap
                        corpusSize
                        avgDocLen
                        k
                        b
       )
     )
     docs
   `using` parListChunk chunks rseq

getDocWithScore
 :: [String]
 -> Document
 -> Map.Map String Int
 -> Int
 -> Int
 -> Double
 -> Double
 -> (Document, Double)
getDocWithScore query doc dfMap corpusSize avgDocLen k b =
 (doc, sum [ individualScore q | q <- query ]) where
 individualScore :: String -> Double
 individualScore queryWord =
   calcIDF queryWord
     * fromIntegral (calcTF queryWord)
     * (k + 1.0)
     / ( fromIntegral (calcTF queryWord)
       + k
       * (1.0 - b + b * fromIntegral corpusSize / fromIntegral avgDocLen)
       )
 calcTF :: String -> Int
 calcTF queryWord = fromMaybe 0 (Map.lookup queryWord (termFrequency doc))
 calcIDF :: String -> Double
 calcIDF queryWord
   | Map.member queryWord dfMap = ln (fromIntegral corpusSize + 1)
```

```
   - ln (fromIntegral (fromMaybe 0 (Map.lookup queryWord dfMap)) + 0.5)
   | otherwise = ln (fromIntegral corpusSize * 2)
   where ln = logBase (exp 1)

readDocument :: String -> String -> Document
readDocument title' text' = Document title'
                                     text'
                                     tfMap
                                     (Map.keysSet tfMap)
                                     (sum (Map.elems tfMap))
 where tfMap = buildTFMap text'

-- Get the document count (DF) for each word
getAllDF :: [Document] -> Map.Map String Int
getAllDF docs = Map.fromListWith
 (+)
 [ tup | doc <- docs, tup <- zip (Set.toList (termSet doc)) [1 :: Int, 1 ..] ]

topKDocuments
 :: Int -> [(Double, Document)] -> Heap.MinPrioHeap Double Document
topKDocuments _ [] = Heap.fromList [] :: Heap.MinPrioHeap Double Document
topKDocuments topK (sd : sds) = maybePop
 (Heap.insert sd (topKDocuments topK sds)) where
 maybePop
   :: Heap.MinPrioHeap Double Document -> Heap.MinPrioHeap Double Document
 maybePop minHeap | Heap.size minHeap > topK = Heap.drop 1 minHeap
                  | otherwise                = minHeap

buildTFMap :: String -> Map.Map String Int
buildTFMap textString =
 Map.fromListWith (+) [ (word, 1) | word <- tokenizeAndNormalize textString ]

-- Return a list of normalized tokens
tokenizeAndNormalize :: String -> [String]
tokenizeAndNormalize query = map sanitizeWord (words query)
 where sanitizeWord word = map toLower (filter isLetter word)
```

---

main.hs

```
import           Control.Monad.Reader
import           Control.Parallel.Strategies
import           Data.Char
import qualified Data.Heap                    as Heap
import           Data.Time
import           OkapiBM25_utils
import           Prelude
import           System.Environment            ( getArgs
                                               , getProgName
                                               )
import           System.Exit                   ( die )

main :: IO ()
main = do
 (file, query, topK, chunks) <- getArgs >>= checkArgs
 output                      <- runOkapiBM25 file
                                             (filterLetters query)
                                             (read topK :: Int)
                                             (read chunks :: Int)
 mapM_ print output
where
 checkArgs [file, query, topK, chunks] = return (file, query, topK, chunks)
 checkArgs _                           = do
   progName <- getProgName
   die
     $  "Usage: ./"
     ++ progName
     ++ " <filename> <query> <K (no. of top documents)> <no. of chunks>"
 filterLetters = filter (\c -> isLetter c || isSpace c || c == '\'')


runOkapiBM25 :: String -> String -> Int -> Int -> IO [(String, Double)]
runOkapiBM25 file query topK chunks = do
 documents' <- getAllDocuments file chunks
 documents  <- sequence documents'
```

```
 let corpusSize = length documents -- N
 let (avgDocLen, dfMap) = runEval $ do
       avgDocLen' <-
         rpar $ (sum [ docLen doc | doc <- documents ]) `div` corpusSize
       dfMap' <- rpar $ getAllDF documents
       _      <- rseq avgDocLen'
       _      <- rseq dfMap'
       return (avgDocLen', dfMap')

 let (k, b) = (1.2, 0.75)
 startTime <- documents `seq` getCurrentTime
 let scoredDocs =
       scoreAllDocs chunks query documents dfMap corpusSize avgDocLen k b
 let result = parseOutput (Heap.toAscList (topKDocuments topK scoredDocs))
 endTime <- result `seq` getCurrentTime
 print $ diffUTCTime endTime startTime
 return result

getAllDocuments :: String -> Int -> IO [IO Document]
getAllDocuments file chunks = do
 fileText <- readFile file -- titles.txt
 let documents =
       [ fmap (\(text', title'') -> readDocument title'' text')
              (readFile path >>= return . \fileTxt -> (fileTxt, title'))
       | line <- lines fileText
       , let (path, title') = splitAtFirst ',' line
       ]
       `using` parListChunk chunks rseq
 return documents
 where splitAtFirst x = fmap (drop 1) . break (x ==)

parseOutput :: [(Double, Document)] -> [(String, Double)]
parseOutput inputs = reverse [ (title doc, score) | (score, doc) <- inputs ]
```
