Anthony Poerio (adp59@pitt.edu)
Andrew Masih (anm226@pitt.edu)
CS1699 – Cloud Computing
Tiny Google – Spark
Final Report

TinyGoogle - Spark

Overview

For our final project, we created a second implementation of the **Tiny Google Search Engine** written to run on the **Spark** Cloud Computing framework, using **Scala** as the implementation language.

Our goals for this document are as follows:

- We will outline the overarching design for our **Tiny Google** search engine.
- We will discuss the implementation details and major decisions made to ensure that Tiny Google runs correctly and efficiently on the Scala/Spark framework.
- We will share output for a series of example keyword searches.
- Finally, we will compare and contrast our original **Python/ Hadoop Streaming** solution with our new **Scala / Spark** implementation, in order to make a determination about which method is preferable – both for programmers and for end-users.

Design

User Input

Similar to our original version of **Tiny Google**, we decided to give the Spark implementation a command line interface.

The interface works as follows:

1. **To INDEX our data**
   a. Pass in the keyword argument **-i**, and the directory containing your dataset
   b. Example: *`scala tinyGoogle.scala –i /my/path/to/books/dataset/`*
2. **To SEARCH an index, using keywords**
   a. Pass in the keyword argument **–s**, and a set of keywords
   b. Example: *`scala tinyGoogle.scala –s "this is an example search"`*

If a user decides to search before an index is created, we will prompt the user to build an index before running a search.

Output is passed to the command line console, and we show the **Top-3 results** for EACH keyword, in order of user-entry.

Indexing Algorithm & Data Structure

In order to make the most of Spark's newest and most optimized features, we decided to use a **DataFrame** to store our index.

For a programmer, the main advantages of this decision are:

- Spark DataFrames are easy to use and make
- They provide a fast and efficient SQL-like querying interface
- And the resultant code is easy to understand and modify

For an end-user the advantage is speed. First introduced in Spark Version 1.6, queries on DataFrames are optimized by Spark's internal code, ensuring that functions are executed in the most efficient order. This allows us to search a created index with high-efficiency – all while offloading the optimizations themselves to Spark.

The chief challenge here is designing an efficient DataFrame format (columns) that provides quick access to all necessary information, using SQL-like queries. A secondary, manipulating data such that it can be parsed by the DataFrame API. And for that RDD's were used. More details on these transformations will be outlined below, in the implementation details section.

We chose the following column format for the DataFrame we created.

```
+--------------------+----------+-----+
|               title|      word|count|
+--------------------+----------+-----+
|                 N/A|starter_df|    1|
|AdventuresOfHuckl...|    widows|    2|
|AdventuresOfHuckl...| reforming|    3|
|AdventuresOfHuckl...|    morals|    2|
|AdventuresOfHuckl...|    county|    2|
|AdventuresOfHuckl...|     doing|   30|
|AdventuresOfHuckl...|     clock|   18|
                and so on
```

With this format in hand, we are able to write simple FILTER and GROUPBY queries at search time, quickly finding results for the end-user.

In fact, Spark is able to optimize our queries, precisely because the data we are searching through is in a DataFrame format.

Search Algorithm & Displaying Results

Our search algorithm works by showing the **TOP 3** relevant books for each search term, ordered **by occurrence counts**.

We achieve this by using the Spark DataFrame API to query the index built in the previous step.

We are also able to display context by finding the FIRST occurrence of the target keyword in any given book, and indexing directly to that line, showing the surrounding data.

Because we have all of this data encoded in our DataFrame index, and because Spark helps us with optimizations for each search query on DataFrames, we are able to perform these queries quickly and easily.

Results from our searches will be discussed and showcased in the “Example Runs” section, below.

Implementations

User Interface

We were able to quickly and easily handle user input, and the overall interface by parsing command line arguments through the ***main(args: Array[String])*** function, just like in standard Java.

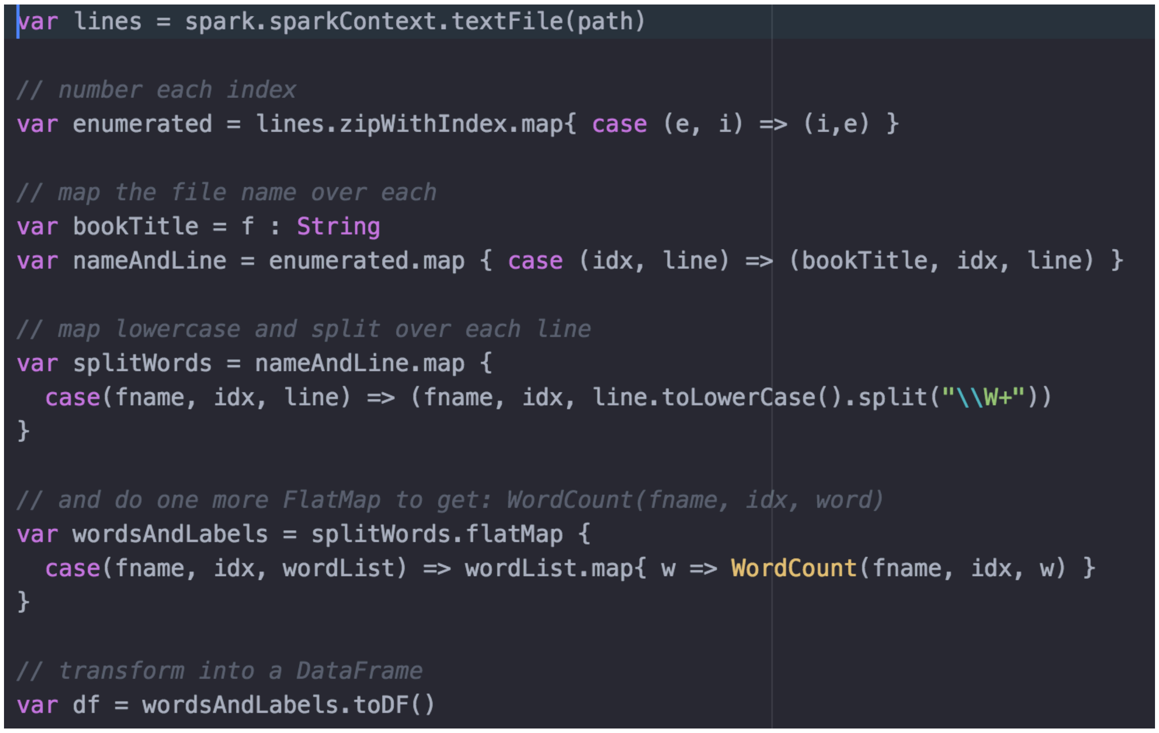
This made reading and handling user-input surprisingly simple. From there, we were able to check and validate input using try/catch blocks, ensuring that the input passed in is valid, and that an index is created before we attempt to search.

At the end of a search query, we were also able to print the results directly to STDOUT, as expected.

Indexing

We indexed our dataset by loading each file from the user's target directory and running it through a series of RDD transformations.

Here's a quick and commented walk through of the RDD transformations we performed, taken from our source code.

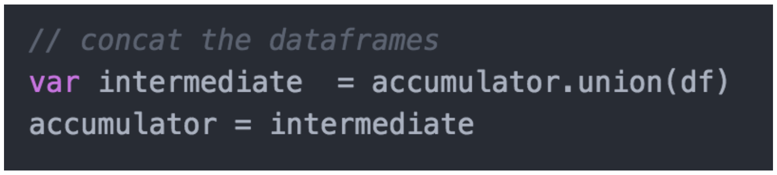


```
var lines = spark.sparkContext.textFile(path)

// number each index
var enumerated = lines.zipWithIndex.map{ case (e, i) => (i,e) }

// map the file name over each
var bookTitle = f : String
var nameAndLine = enumerated.map { case (idx, line) => (bookTitle, idx, line) }

// map lowercase and split over each line
var splitWords = nameAndLine.map {
  case(fname, idx, line) => (fname, idx, line.toLowerCase().split("\\W+"))
}

// and do one more FlatMap to get: WordCount(fname, idx, word)
var wordsAndLabels = splitWords.flatMap {
  case(fname, idx, wordList) => wordList.map{ w => WordCount(fname, idx, w) }
}

// transform into a DataFrame
var df = wordsAndLabels.toDF()
```

Once we have an RDD in a format that is roughly similar to our DataFrame's column design, we pass each tuple within the RDD to a **Scala Case Class.** This is done in the ***FlatMap*** step, above. The purpose of this step is to ensure that the DataFrame API is able to parse all of our data in in a consistent way.

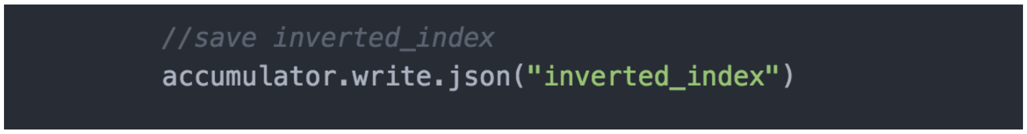
Finally, we perform this same set of operations for EACH target file, and union the resulting DataFrame with an accumulator, which holds data for the entire directory.

Here's where the union takes place.



```
// concat the dataframes
var intermediate  = accumulator.union(df)
accumulator = intermediate
```

Once these operations are complete, we save the index in JSON format, using functions built into Spark directly.



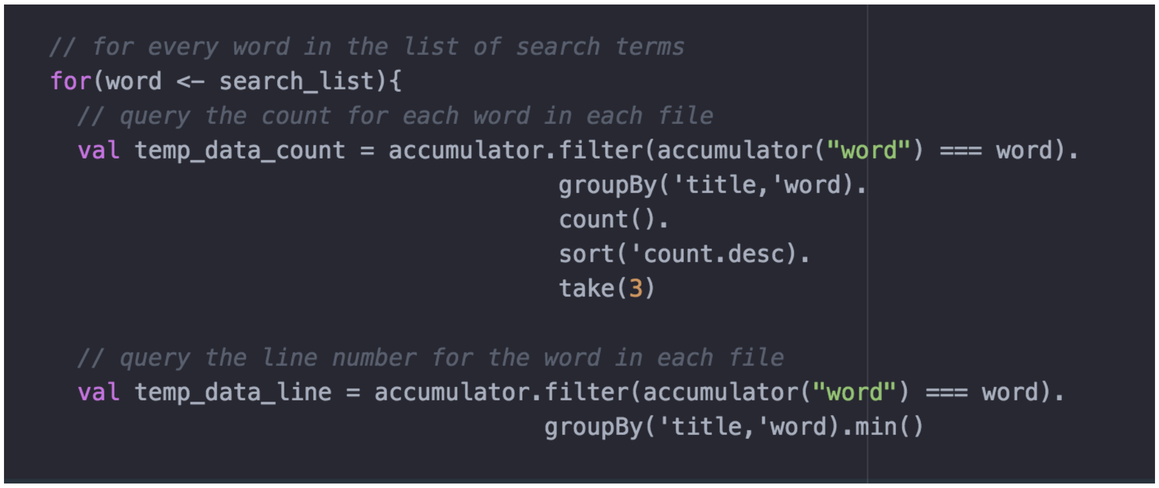
```
        //save inverted_index
        accumulator.write.json("inverted_index")
```

Because we save with a built-in Spark function, we are able to load the data from this format quickly and easily—using one command at search time.

Searching

Our search algorithm performs two actions. First, it finds the Top 3 Books for each keyword, sorted by occurrences. Then, for context, we find the very first appearance of the word in each book. This data is used to provide the user with an example 'context', showing how the word is used in each book.

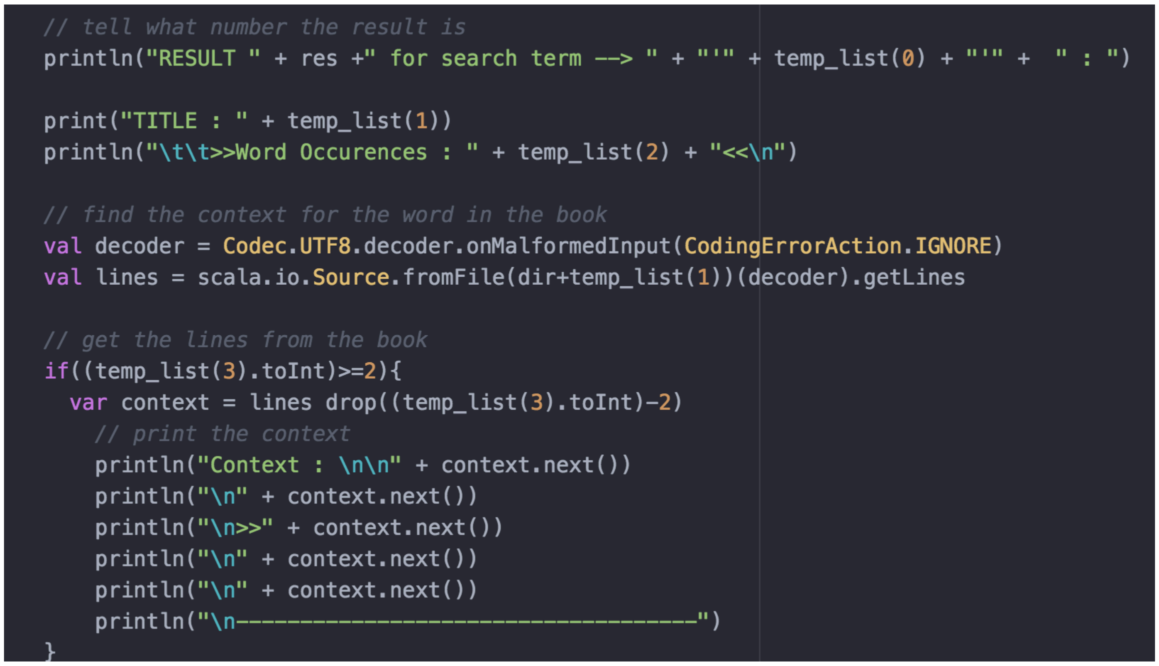
For the first action—where we find the Top 3 Books—we were able to implement this fully using queries on the DataFrame API. This is the pertinent code, showing how this is achieved.



```
// for every word in the list of search terms
for(word <- search_list){
  // query the count for each word in each file
  val temp_data_count = accumulator.filter(accumulator("word") === word).
                                   groupBy('title,'word).
                                   count().
                                   sort('count.desc).
                                   take(3)

  // query the line number for the word in each file
  val temp_data_line = accumulator.filter(accumulator("word") === word).
                                  groupBy('title,'word).min()
```

Here, we can see that by using combinations of Filter and GroupBy, on each of our DataFrame columns, we are able to gather the top 3 books using simple and straightforward SQL-like logic.

With our data in hand, next we need to find the context. This is achieved in the following code snippet. We find the context by using the line numbers from the data found in the previous step. Using these line numbers, we can read in each file and index to the proper line directly.



```scala
// tell what number the result is
println("RESULT " + res +" for search term --> " + "'" + temp_list(0) + "'" +  " : ")

print("TITLE : " + temp_list(1))
println("\t\t>>Word Occurences : " + temp_list(2) + "<<\n")

// find the context for the word in the book
val decoder = Codec.UTF8.decoder.onMalformedInput(CodingErrorAction.IGNORE)
val lines = scala.io.Source.fromFile(dir+temp_list(1))(decoder).getLines

// get the lines from the book
if((temp_list(3).toInt)>=2){
  var context = lines drop((temp_list(3).toInt)-2)
    // print the context
    println("Context : \n\n" + context.next())
    println("\n" + context.next())
    println("\n>>" + context.next())
    println("\n" + context.next())
    println("\n" + context.next())
    println("\n-----------------------------------")
}
```

We also need to handle errors, such as converting Unicode values to UTF-8. But once that is handled, we are able to easily display the final result to our end-users.

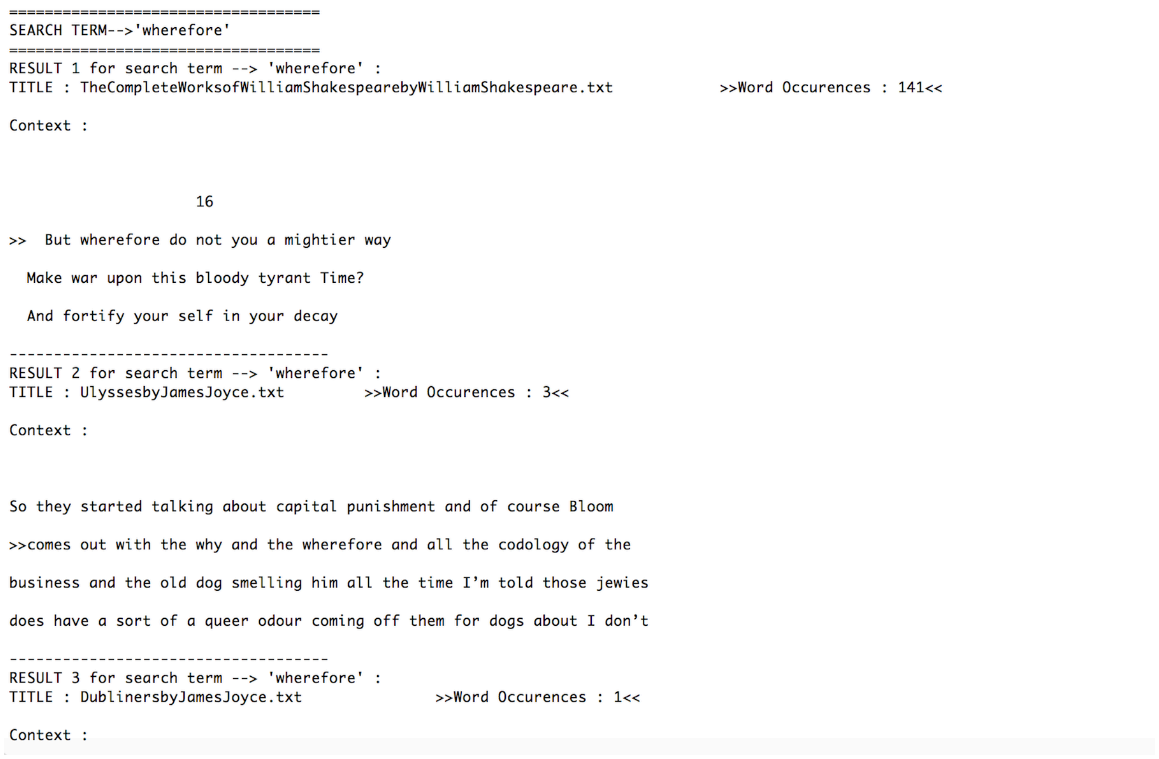
Example Runs

Each of these example runs show the top results. For full output (as a text-file), please see the **/example_runs** folder. The results for each search are labeled accordingly.

"bingley"

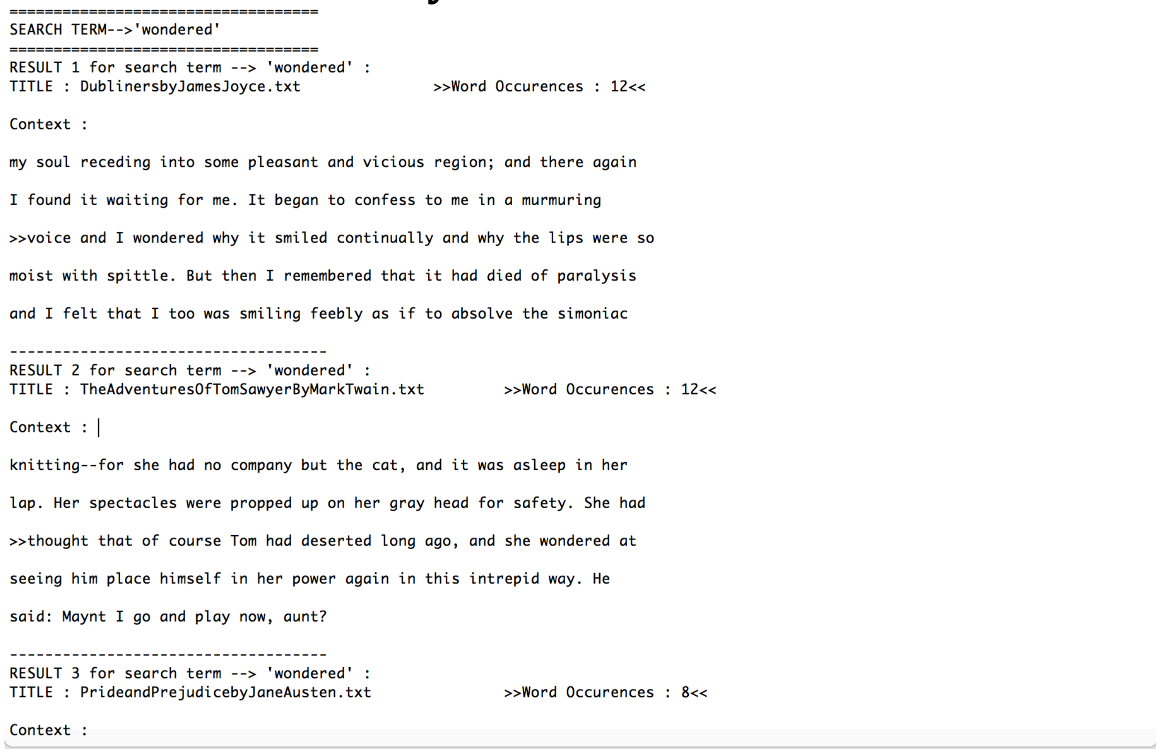
```
===================================
SEARCH TERM-->'bingley'
===================================
RESULT 1 for search term --> 'bingley' :
TITLE : PrideandPrejudicebyJaneAusten.txt               >>Word Occurences : 306<<

Context :

"What is his name?"



>>"Bingley."



"Is he married or single?"

-----------------------------------
```

“wherefore art thou romeo”



```
===================================
SEARCH TERM-->'wherefore'
===================================
RESULT 1 for search term --> 'wherefore' :
TITLE : TheCompleteWorksofWilliamShakespearebyWilliamShakespeare.txt           >>Word Occurences : 141<<

Context :



                     16

>>  But wherefore do not you a mightier way

  Make war upon this bloody tyrant Time?

  And fortify your self in your decay

------------------------------------
RESULT 2 for search term --> 'wherefore' :
TITLE : UlyssesbyJamesJoyce.txt         >>Word Occurences : 3<<

Context :


So they started talking about capital punishment and of course Bloom

>>comes out with the why and the wherefore and all the codology of the

business and the old dog smelling him all the time I’m told those jewies

does have a sort of a queer odour coming off them for dogs about I don’t

------------------------------------
RESULT 3 for search term --> 'wherefore' :
TITLE : DublinersbyJamesJoyce.txt               >>Word Occurences : 1<<

Context :
```

“wondered uneasily”



```
===================================
SEARCH TERM-->'wondered'
===================================
RESULT 1 for search term --> 'wondered' :
TITLE : DublinersbyJamesJoyce.txt               >>Word Occurences : 12<<

Context :

my soul receding into some pleasant and vicious region; and there again

I found it waiting for me. It began to confess to me in a murmuring

>>voice and I wondered why it smiled continually and why the lips were so

moist with spittle. But then I remembered that it had died of paralysis

and I felt that I too was smiling feebly as if to absolve the simoniac

------------------------------------
RESULT 2 for search term --> 'wondered' :
TITLE : TheAdventuresOfTomSawyerByMarkTwain.txt         >>Word Occurences : 12<<

Context : |

knitting--for she had no company but the cat, and it was asleep in her

lap. Her spectacles were propped up on her gray head for safety. She had

>>thought that of course Tom had deserted long ago, and she wondered at

seeing him place himself in her power again in this intrepid way. He

said: Maynt I go and play now, aunt?

------------------------------------
RESULT 3 for search term --> 'wondered' :
TITLE : PrideandPrejudicebyJaneAusten.txt               >>Word Occurences : 8<<

Context :
```

Comparison Analysis

Ease of Implementation

Overall, the same tasks can be accomplished with both Python/Hadoop Streaming, and Scala/Spark. But using Scala/Spark, the implementation much more compact.

In Python it took us 574 lines of code—and we used multiple Bash shell scripts, to tie everything together.

In Scala, we only needed 245 lines of code, and everything was self contained in a single file.

Overall, the Scala code lets you interact with your data structures from a much more abstract and high-level perspective. You focus more on queries and tasks, instead of low level implementation details.

For this reason, we believe that implementing a Tiny Google is much simpler using Scala/Spark. The hardest part of Spark is really getting the IDE and development environment setup. Once you're past that, initial hurdle, you can iterate much more quickly.

Python MapReduce

inverted-index = 35.6 (sec)

search term : bingley = 36.37 (sec)

search term : wondered uneasily = 37.47 (sec)

search term : wherefore art thou romeo = 39.61 (sec)

Scala Spark

inverted-index = 14.47 (sec)

search term : bingley = 19.60 (sec)

search term : wondered uneasily = 33.17 (sec) search term : wherefore art thou romeo = 53.93 (sec)

Conclusions

Pros and Cons

The implementation is simpler using Scala. But getting the environment setup is easier in Python.

Creating an inverted index was about 2x faster in Scala. And in general, search times were shorter.

However, in our Scala implementation, search time is a function of keyword length. As the keywords get longer, we must perform additional queries, and therefore, it takes longer to search.

This means that for very long queries, our Python solution would be faster.

Final Decision

We believe that, overall, Scala/Spark is a better tool for this job. Given the opportunity to code a full-scale system, we believe that any limitations in our naïve version can be overcome with the great tools available in Spark.

Writing higher-level Scala code (and doing the same thing in fewer lines) means that there is less code to maintain long-term, and thus the code base is simpler to understand and more maintainable.

Therefore, we would prefer the Spark implementation to the Python implementation in this case.