Practice Programming Assignment: Wikipedia

You have not submitted. You must earn 8/10 points to pass.

Deadline Pass this assignment by December 24, 11:59 PM PST

Instructions My submission Discussions

Note: If you have paid for the Certificate, please make sure you are submitting to the required assessment and not the optional assessment. If you mistakenly use the token from the wrong assignment, your grades will not appear

How to submit

Copy the token below and run the submission script included in the assignment download. When prompted, use your email address **serg.astapov@gmail.com**.

PPZtdHXLOFtmCy2z

Generate new token

Your submission token is unique to you and

Wikipedia

To start, first download the assignment: wikipedia.zip. For this assignment, you also need to download the data (133 MB):

http://alaska.epfl.ch/~dockermoocs/bigdata/wikipedia.dat

and place it in the folder: **src/main/resources/wikipedia** in your project directory.

In this assignment, you will get to know Spark by exploring full-text Wikipedia articles.

Gauging how popular a programming language is important for companies judging whether or not

they should adopt an emerging programming language. For that reason, industry analyst firm RedMonk has bi-annually computed a ranking of programming language popularity using a variety of data sources, typically from websites like GitHub and StackOverflow. See their top-20 ranking for June 2016 as an example.

In this assignment, we'll use our full-text data from Wikipedia to produce a rudimentary metric of how popular a programming language is, in an effort to see if our Wikipedia-based rankings bear any relation to the popular Red Monk rankings.

You'll complete this exercise on just one node (your laptop), but you can also head over to Databricks Community Edition to experiment with your code on a "micro-cluster" for free.

Set up Spark

For the sake of simplified logistics, we'll be running Spark in "local" mode. This means that your full Spark application will be run on one node, locally, on your laptop.

To start, we need a **SparkContext**. A **SparkContext** is the "handle" to your cluster. Once you have a **SparkContext**, you can use it to create and populate RDDs with data.

To create a **SparkContext**, you need to first create a **SparkConfig** instance. A **SparkConfig** represents the configuration of your Spark application. It's here that you must specify that you intend to run your application in "local" mode. You must also name your Spark application at this point. For help, see the Spark API Docs.

Configure your cluster to run in local mode by implementing **val conf** and **val sc**.

Read-in Wikipedia Data

should not be shared with anyone. You may submit as many times as you like.

There are several ways to read data into Spark. The simplest way to read in data is to convert an existing collection in memory to an RDD using the **parallelize** method of the Spark context.

We have already implemented a method **parse** in the object **WikipediaData** object that parses a line of the dataset and turns it into a **WikipediaArticle**.

Create an RDD (by implementing **val wikiRdd**) which contains the **WikipediaArticle** objects of **articles**.

Compute a ranking of programming languages

We will use a simple metric for determining the popularity of a programming language: the number of Wikipedia articles that mention the language at least once.

Rank languages attempt #1: rankLangs

Computing occurrencesOfLang

Start by implementing a helper method **occurrencesOfLang** which computes the number of articles in an **RDD** of type **RDD[WikipediaArticles]** that mention the given language at least once. For the sake of simplicity we check that it least one word (delimited by spaces) of the article text is equal to the given language.

Computing the ranking, rankLangs

Using **occurrencesOfLang**, implement a method **rankLangs** which computes a list of pairs where the second component of the pair is the number of articles that mention the language (the first component of the pair is the name of the language).

An example of what **rankLangs** might return might look like this, for example:

```
List(("Scala", 999999), ("JavaScript", 1278), ("LOLCODE", 982),
    ("Java", 42))
```

The list should be sorted in descending order. That is, according to this ranking, the pair with the highest second component (the count) should be the first element of the list.

Pay attention to roughly how long it takes to run this part! (It should take tens of seconds.)

Rank languages attempt #2: rankLangsUsingIndex

Compute an inverted index

An inverted index is an index data structure storing a mapping from content, such as words or numbers, to a set of documents. In particular, the purpose of an inverted index is to allow fast full text searches. In our use-case, an inverted index would be useful for mapping from the names of programming languages to the collection of Wikipedia articles that mention the name at least once.

To make working with the dataset more efficient and more convenient, implement a method that computes an "inverted index" which maps programming language names to the Wikipedia articles on which they occur at least once.

Implement method **makeIndex** which returns an RDD of the following type: **RDD[(String, Iterable[WikipediaArticle])]**. This RDD contains pairs, such that for each language in the given **langs** list there is at most one pair. Furthermore, the second component of each pair (the **Iterable**) contains the **WikipediaArticles** that mention the language at least once.

*Hint: You might want to use methods **flatMap** and **groupByKey** on **RDD** for this part.*

Computing the ranking, rankLangsUsingIndex

Use the **makeIndex** method implemented in the previous part to implement a faster method for computing the language ranking.

Like in part 1, **rankLangsUsingIndex** should compute a list of pairs where the second component of the pair is the number of articles that mention the language (the first component of the pair is the name of the language).

Again, the list should be sorted in descending order. That is, according to this ranking, the pair with the highest second component (the count) should be the first element of the list.

*Hint: method **mapValues** on **PairRDD** could be useful for this part.*

Can you notice a performance improvement over attempt #2? Why?

Rank languages attempt #3: rankLangsReduceByKey

In the case where the inverted index from above is only used for computing the ranking and for no other task (full-text search, say), it is more efficient to use the **reduceByKey** method to compute the ranking directly, without first computing an inverted index. Note that the **reduceByKey** method is only defined for RDDs containing pairs (each pair is interpreted as a key-value pair).

Implement the **rankLangsReduceByKey** method, this time computing the ranking without the inverted index, using **reduceByKey**.

Like in part 1 and 2, **rankLangsReduceByKey** should compute a list of pairs where the second component of the pair is the number of articles that mention the language (the first component of the pair is the name of the language).

Again, the list should be sorted in descending order. That is, according to this ranking, the pair with the highest second component (the count) should be the first element of the list.

Can you notice an improvement in performance compared to measuring both the computation of the index and the computation of the ranking as we did in attempt #2? If so, can you think of a reason?