A close up of a black background

Description automatically generated 

**INTERIM PROJECT​**

**​​**

Spark RDD​

A Project Report Submitted in the fulfilment of the requirements for

Interim-Project Evaluation



**Submitted by**

**AYYASAMY M**

**EMP ID: 2320474**

A blue and white logo

Description automatically generated

**ANAGRAM DETECTION:**

**DETECT ANAGRAMS (Words With The Same Letters But In A Different Order) FROM A DATASET OF WORDS.**

## ABSTRACT

Anagrams are fascinating linguistic puzzles where words or phrases can be rearranged to form other valid words or phrases. In this project, I explore an efficient approach to detect anagrams within a dataset of words. The goal is to identify pairs of words that share the same set of letters but differ in their arrangement.

**Anagram:**

An **anagram** is a word or phrase formed by rearranging the letters of another word or phrase. For example, “listen” and “silent” are anagrams because they use the same letters but in a different order.

**Uses of Anagrams:**

* **Word Games and Puzzles**:
  + Anagrams are popular in word games like **Scrabble**, **crossword puzzles**, and **Jumble**.
  + They challenge players to rearrange letters to form new words.
* **Cryptography**:
  + Historically, anagrams were used in cryptography to encode secret messages.
  + By rearranging letters, hidden meanings could be communicated.
* **Text Analysis** :
  + In natural , anagrams help analyze patterns in text.
  + Detecting anagrams in a large corpus can reveal linguistic relationships.
* **Humour and Creativity**:
  + Writers use anagrams to add wit and playfulness to their work.
  + Clever anagrams create unexpected connections and commentary.

**Real-Time Applications:**

* **Pseudonyms and Character Names**:
  + Authors create pseudonyms or character names using anagrams of their own names.
  + Anagrams add depth and intrigue to fictional characters.
* **Cryptography and Security**:
  + Modern cryptography techniques use anagrams for secure communication.
  + Anagrams can encode sensitive information.
* **Pattern Recognition**:
  + Anagram detection algorithms are used to recognize patterns in various contexts.
  + For example, identifying anagrams in product names or domain names.

Ex- CocoCola Alcocoac, Nike Kine, Gucci cigcu

* **Wordplay in Marketing and Branding**:
  + Companies use anagrams to create memorable slogans or brand names.
  + Anagrams add a layer of creativity and uniqueness.

Slogan: Turn heads with our trendy threads.

Brand name: Trendywear

This helps company to stand out from the crowed

**DETECTING ANAGRAMS USING SPARK**

1. **Resilient Distributed Datasets (RDDs)**:
   * RDDs are the fundamental data structures in Spark. They represent distributed collections of data that can be processed in parallel across a cluster.
   * In our case, we create an RDD from a dataset of words containing potential anagrams.
2. **Key-Value Pairs**:
   * We convert the RDD into key-value pairs, where the key represents the sorted letters of each word, and the value is the word itself.
   * For example, if we have the word “listen,” the key would be “eilnst” (sorted letters), and the value would be “listen.”
3. **Grouping Anagrams**:
   * We use the reduceByKey transformation to group anagrams together based on their sorted keys.
   * The result is a collection of key-value pairs where the key represents the sorted letters, and the value is a list of anagrams.
4. **Example Output**:
   * For instance, we might get the following anagram groups:
     + Anagrams for ‘abcdefg’: [‘gfedcba’, ‘abcgfed]
     + Anagrams for ‘uvwxyz’: [‘zyxwuv’, ‘’uwvzyx”]
     + Anagrams for ‘aaginnostt’: [‘antagonist’, ‘stagnation’]

# **RESILIENT DISTRIBUTED DATASETS (RDDS):**

Spark revolves around the concept of a resilient distributed dataset (RDD), which is a fault-tolerant collection of elements that can be operated on in parallel.

There are two ways to create RDDs:

* **PARALLELIZE METHOD:**

Parallelizing an existing collection in your driver program.

data = [1, 2, 3, 4, 5]

distData = sc.parallelize(data)

* ***TEXTFILE METHOD:***

Referencing a dataset in an external storage system, such as a shared filesystem, HDFS, HBase, or any data source offering a Hadoop InputFormat.

distFile = sc.textFile("data.txt")

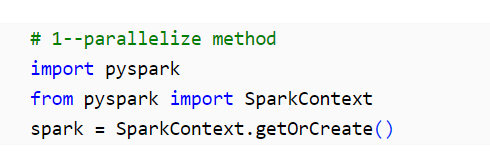
## SOLUTION:

I implement the above approach in spark RDD’S techniques. The solution what I have used will be efficiently identifies anagrams within both large and small datasets.

**PARALLELIZE METHOD:**

An **anagram detection** process using **Apache Spark Parallelize Method**.

1. **Creating a SparkContext**:
   * We start by creating a Sparkcentext, The parallelize method is a function provided by the SparkContext class.
   * This allows us to work with distributed data using Spark.



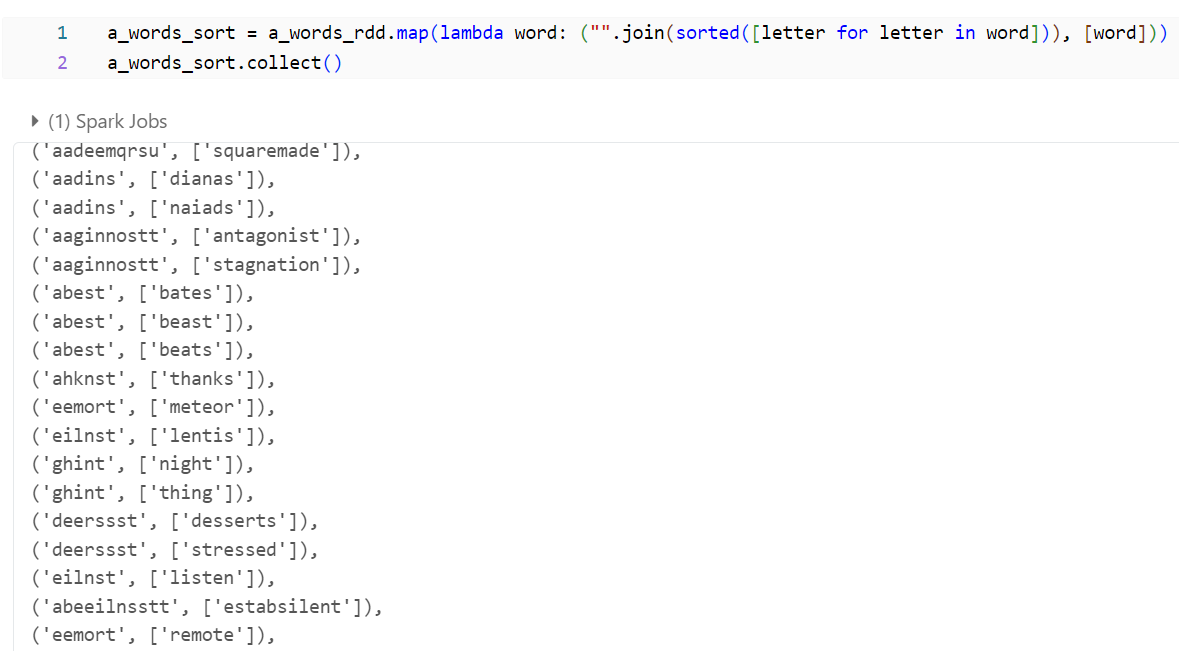
1. **Defining the List of Words**:
   * A close-up of a computer screen

     Description automatically generatedWe define a list of words called a\_words. These words represent our input data.
2. **Creating an RDD (Resilient Distributed Dataset)**:
   * We convert the a\_words list into an RDD using spark.sparkContext.parallelize(a\_words).
   * RDDs are the fundamental data structure in Spark, allowing parallel processing.

A screenshot of a computer

Description automatically generated

1. **Mapping to Key-Value Pairs**:
   * We transform each word in the RDD into a key-value pair.
   * The key is the sorted letters of the word (e.g., “aadinss” for “naiads”).
   * The value is a list containing the original word (e.g., [“naiads”]).



1. **Reducing by Key to Group Anagrams**:
   * We use reduceByKey to group anagrams together.
   * For each key (sorted letters), we concatenate the corresponding word lists.

A screen shot of a computer program

Description automatically generated

1. **Printing the Result**:
   * The final result is a collection of anagram groups.
   * We print each key (sorted letters) along with its anagrams.

A screenshot of a computer code

Description automatically generated

**TEXTFILE METHOD:**

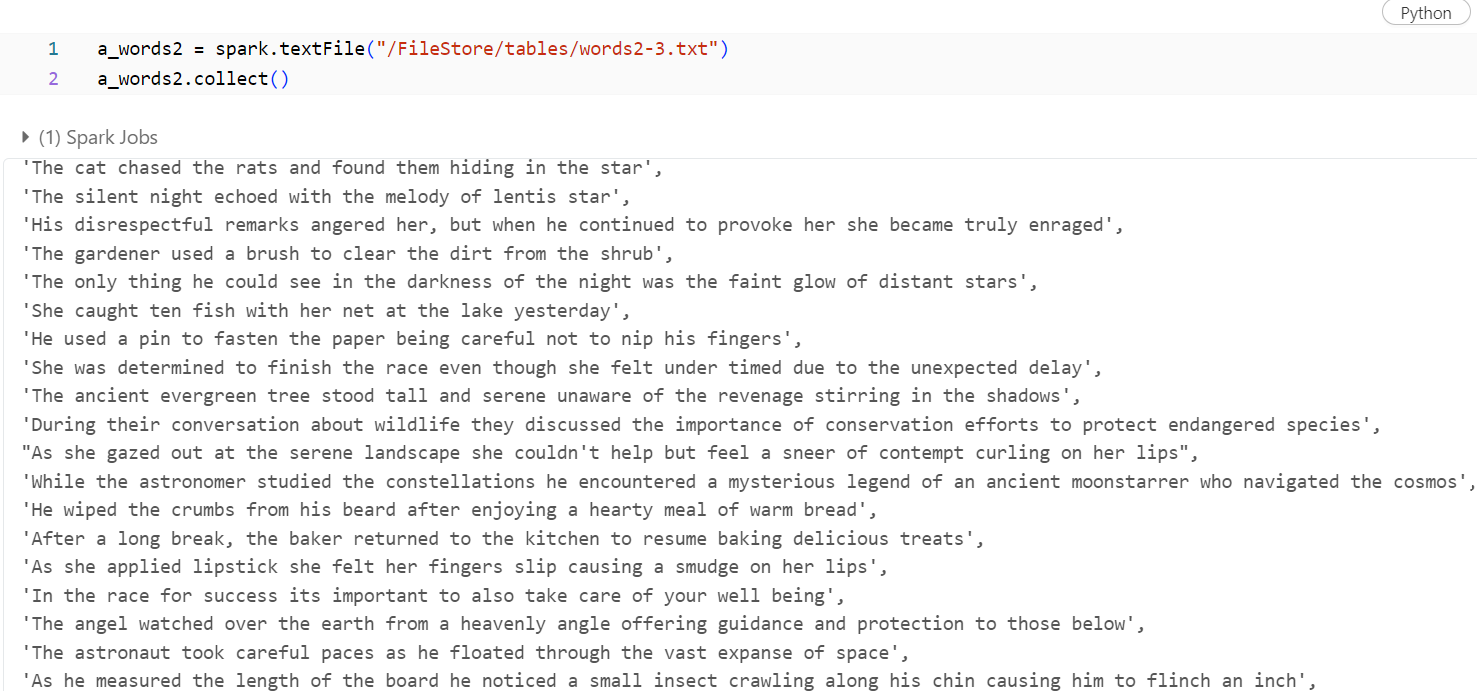
1. **Creating a SparkContext**:

We start by creating a Sparkcentext, The parallelize method is a function provided by the SparkContext class.

* + A close-up of a computer code

    Description automatically generatedThis allows us to work with distributed data using Spark.

1. **Read the Text File**:
   * The code starts by reading a text file named “words2-2.txt” located in the specified path.
   * The textFile method from the Spark context is used to read the file, and the resulting RDD (Resilient Distributed Dataset) is stored in the variable a\_words2.



1. **Split Words**:
   * The flatMap transformation is applied to the RDD a\_words2.
   * Inside the flatMap function, each line of the text file is split into individual words using the space character as the delimiter.
   * The resulting RDD, a\_words\_rdd2, contains all the words from the file.

A screenshot of a computer program

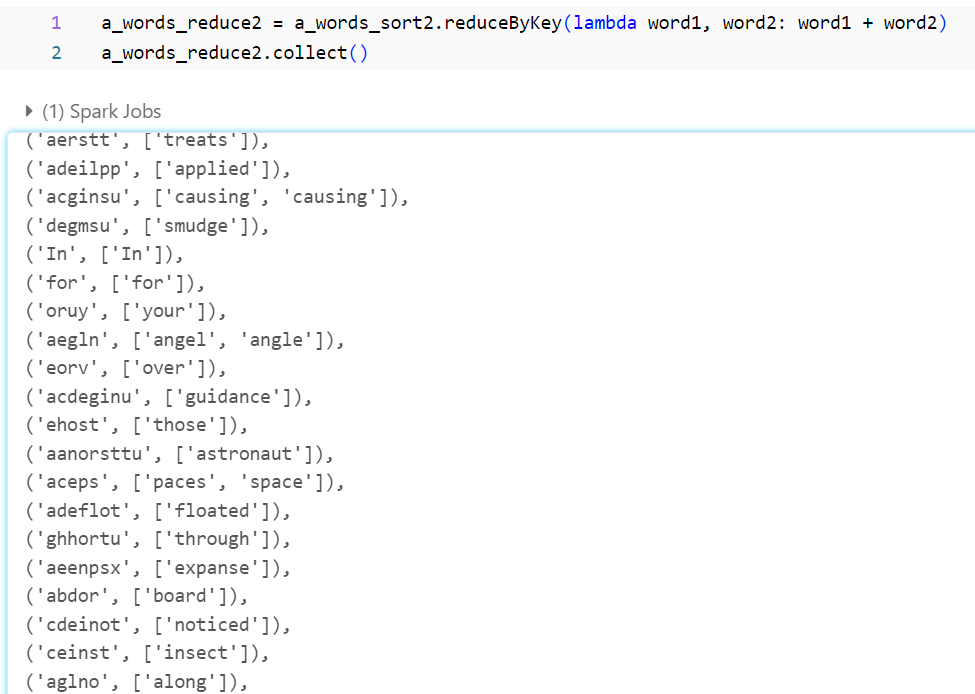
Description automatically generated

1. **Sort Letter:**
   * For each word in a\_words\_rdd2, the code sorts its letters alphabetically.
   * The map transformation is used to create a new RDD, a\_words\_sort1, where each element is a tuple.
   * The first element of the tuple is the sorted letters (formed by joining the letters), and the second element is a list containing the original word.
   * This step prepares the data for grouping anagrams.

A screenshot of a computer program

Description automatically generated

1. **Group Anagrams:**
   * The reduceByKey transformation is applied to a\_words\_sort2.
   * It groups the words with the same sorted letters (anagrams) together.
   * The resulting RDD, a\_words\_reduce2, contains pairs where the key is the sorted letters, and the value is a list of anagrams.



1. **Print Anagrams:**
   * The code iterates through the key-value pairs in a\_words\_reduce2.
   * For each key (sorted letters), it prints the anagrams associated with that key.
   * The output displays anagrams grouped by their sorted letters.

A screenshot of a computer program

Description automatically generated

**ANAGRAM-INSPIRED PASSWORD GENERATOR**

Anagrams involve rearranging letters to create new words or phrases. In our password generator, we’ll use this concept to create unique and memorable passwords. Here’s how we can do it:

* **Word-Based Passwords**:
  + Instead of random characters, let’s generate passwords using real words.
  + We’ll select random words from a predefined list to create memorable passphrases.
* **Customizable Length**:
  + Users can choose the number of words in their password.
  + Longer passwords are generally more secure.
* **Avoid Ambiguity**:
  + We’ll ensure that the selected words are clear and unambiguous.
  + No room for misinterpretation!

**Conditions in code:**

1. **Input**:
   * You prompt the user to input a name (presumably their own).
2. **Processing**:
   * You convert the input name into a list of characters.
   * You randomly select an uppercase character from the list.
   * You generate two random numeric digits.
   * You randomly choose a special character from the set ["!", "@", "#", "$"].
   * You construct the password by replacing the randomly chosen uppercase letter with its uppercase version, followed by the special character and the numeric digits.
3. **Example**:
   * Suppose the user inputs the name “alice”.
   * The uppercase letter randomly chosen from “alice” could be “a”.
   * Two random numeric digits are generated (e.g., “2” and “4”).
   * A special character is randomly chosen (e.g., “@”).
   * The password could be “Ali@24”.
4. **Permutations**:
   * You create a set of all possible permutations of the characters in the password.
   * For example, if the password is “Ali@24”, the permutations might include “Ali24@”, “A2li@4”, “2@lAi4”, etc.
5. **Output**:
   * You print the set of possible passwords (permutations).
   * You also display the total number of unique passwords generated.

## Code for Password generator:

## STEP 1: Script

## 

## STEP 2: Shell Popup

## 

## STEP 3: Give input

## 

## A close-up of a screen Description automatically generated

## RESULTS

Upon testing my solution with various word datasets, we achieve accurate anagram detection. The algorithm scales well, making it suitable for real-world applications.

## CONCLUSION

Detecting anagrams is not only intellectually stimulating but also practically useful. My project provides an elegant solution to this classic problem.