**МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ**
**ОБНИНСКИЙ ИНСТИТУТ АТОМНОЙ ЭНЕРГЕТИКИ - филиал**
Федерального государственного автономного образовательного учреждения
высшего образования
**«Национальный исследовательский ядерный университет «МИФИ»**
**(ИАТЭ НИЯУ МИФИ)**

**Кафедра «Информационных систем»**

# ОТЧЕТ

По лабораторной работе №3

## «Разбор и анализ литературного произведения на Apache Spark»

Выполнил: студент группы ИС-М16
Рябов П. В.

Проверил: Грицюк С. В.
Бобков И.А

Обнинск – 2016 г.

В данной лабораторной работе разрабатывалась программа в интегрированной среде разработки Scala IDE, в соответствии с вариантом.

**Описание задания (Вариант** 9: Лев Николаевич Толстой – Воскресение)

Программа должна предоставлять следующий функционал:.

1. Очистка текста;
2. WordCount по тексту;
3. Топ50 наиболее и наименее часто встречающихся слов;
4. Стемминг для слов;
5. Топ50 наиболее и наименее часто встречающихся слов по стемминг.

Для выполнения задачи был использован функциональный подход а цепочка преобразований данных выполнялась на протяжении всего цикла выполнения используя соответствующие выходные и входные значения функций с типом RDD. Для реализации данных методик и выполнения условия масштабируемости был использован Apache Spark в standalone режиме, с предварительной сборкой соответствующих исходных кодов через утилиту-сборщик для java Maven. Для этого в pom.xml были внесены следующие записи в дерево зависимостей проекта:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>

  <groupId>ru.apache.maven</groupId>

  <artifactId>LabThreeProject</artifactId>

  <name>LABA_3</name>

<dependencies>

  <dependency>

    <groupId>org.apache.spark</groupId>

    <artifactId>spark-core_2.11</artifactId>

    <version>1.6.2</version>

  </dependency>

</dependencies>

    <version>3.0</version>

</project>
```

Были выделены отдельно функции очистки текста, для отбрасывания знаков препинания. Данный метод был реализован с помощью.

**Функционал приложения:**

1) Вспомогательные функции печати RDD данных. Отличия лишь в типе принимаемых значений:

```
def printRDD(counts: RDD[(String, Int)], n: Int) {
    counts.take(n).foreach(println)
  }

def printSRDD(counts: RDD[String], n: Int) {
    counts.take(n).foreach(println)
 }
```

2) Получение контекста Spark (под Linux)

```
  def getSparkContext_Linux(): SparkContext = {
    val conf = new   SparkConf().setAppName("ThreeLab").setMaster("local")
    return new SparkContext(conf)
  }
```

Для удобства работы с логами был также установлен уровень логгирования после получения Spark Context: `LogManager.getRootLogger().setLevel(Level.WARN)`

3.1) Функция для удаления служебных символов из набора слов. Очистка производится путем замены всех найденных символов, попадающих под специальное регулярное выражение на пустые символы. Таким образом, на выходе получается очищенный набор слов.

```
def removePunct(words: RDD[String]): RDD[String] = {
    val regex = """[\p{Punct}]"""
    return words.map(_.replaceAll(regex, ""))
  }
```

3.2) Функция получения стоп-листа. Кроме служебных символов необходимо также очистить текст от часто употребляемых слов. Для этого была произведена загрузка соответствующего стоп-листа для русского язык в распределенный набор данных:

```
def getStopList(fileName: String): Array[String] = {
    val stopList = Source.fromURL(getClass.getResource(fileName))
```

```
    return stopList.getLines().toArray

  }
```

4) Подсчет слов. Для получения частот вхождения слов в исходном тексте первоначально все произведение было загружено в RDD контейнер стандартными средствами чтения из текстового файла Spark. Стоп лист загружается как ресурс, из-за небольшого количества требуемой памяти для его загрузки. Далее из полученного RDD набора извлекаются токены по разделителю “ ”, и они же являются словами. На них применяется ранее описанная функция очистки от служебных символов. Далее полученный набор слов обрабатывается на предмет вхождения слов в стоп лист. Соответствующие слова фильтруются функцией filter, а именно:

```
 val cleanWords = words.filter(x
=> !(getStopList(stopListName).contains(x.toLowerCase)))
```

Нижний регистр применяется во избежания дублирования слов с большой и маленькой буквой при проверке. То есть не приходится увеличивать размер стоп-листа в 2 раза.

```
После этого, считаются астоты вхождения через создания кортежа по ключу - слово,
а значение - частота с первоначальным значением единица. Далее происходит
агрегирование по ключу, таким образом количество вхождений слова в тексте
записывается в поле value в паре (Key, Value):
```

```
 val wordCounts = cleanWords.map(word => (word, 1)).reduceByKey(_ + _)
```

Функция запуска Word Count выглядит следующим образом:

```
 def runWordCount(sc: SparkContext, inputPath: String, stopListName: String):
RDD[(String, Int)] = {

    printStopList(stopListName)

    println("=== WordCount Started... ===")

    val text = sc.textFile(inputPath)

    val words = removePunct(text.flatMap(line => line.split(" ")))

    val cleanWords = words.filter(x => !(getStopList(stopListName)

      .contains(x.toLowerCase)))

    val wordCounts = cleanWords.map(word => (word, 1)).reduceByKey(_ + _)

    printRDD(wordCounts, 100)

    //wordCounts.foreach(println)

    return wordCounts

  }
```

5) Вывод Топ и Антитоп 50 по частоте слов. Данная операция выполняется через функции сортировки:

```
def getTop(counts: RDD[(String, Int)], n: Int): Array[(String, Int)] = {
    println("\n=== Top " + n.toString() + "... ===")
    val top = counts.sortBy(_._2, ascending = false)
    return top.take(n)
  }

  def getAntiTop(counts: RDD[(String, Int)], n: Int): Array[(String, Int)]
= {
    println("\n=== AntiTop " + n.toString() + "... ===")
    val top = counts.sortBy(_._2, ascending = true)
    return top.take(n)
  }
```

6) Стемминг. Для выделения стемов из слов использовалась сторонняя библиотека SnowBall.

На её основе была реализована функция возвращающая стем из входного слова:

```
def stemWord(stemmer: russianStemmer, word: String): String = {
    stemmer.setCurrent(word)
    stemmer.stem()
    stemmer.getCurrent()
  }
```

На базе этой функции также был реализован стемминг по набору слов:

```
def stemWords(words: RDD[(String, Int)]): RDD[(String,(String,Int))] = {
    return words.map(x => (stemWord(stemmer, x._1), x))
  }
```

В полном виде функция обрабатывает выход Word Count, проводит стемминг и возращает данные для последующего Word Count по стемам. На печать выводится кортеж состоящий из стема, списка всех слов для которых совпадает стем и суммарное количество вхождений слова в тексте:

```
def runStemmer(sc: SparkContext, data: RDD[(String, Int)]): RDD[(String,
Int)] = {

    println("\n=== Stemming Started... ===")

    val words = data.map(x => x._1)

// Transform (word, N) -> (stem, (word, N))

    val stems = stemWords(data)

    stems.take(100).foreach(println)

// Transform (stem, (word, N)) -> (stem, List[(word, N)])

    val listStems = stems.groupByKey().map(x =>

      (x._1, x._2.map(y => y._1).toList, x._2.map(y => y._2).sum)

    )

    listStems.take(50).foreach(println)


    println("\n=== Extend Table... ===")

    //fullRDD.take(100).foreach(println)

    val stemRDD = stems.map(stems => (stems._1 , 1)).reduceByKey(_ + _)

    return stemRDD

    }
```

Краткая схема функциональных связей: