SVEUČILIŠTE U ZAGREBU
**FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA**

ZAVRŠNI RAD br. 4573

# Obrada podataka tehnologijom Apache Spark

Martin Matak

Zagreb, lipanj 2016.

*Umjesto ove stranice umetnite izvornik Vašeg rada.*
*Da bi ste uklonili ovu stranicu obrišite naredbu* `\izvornik`. Na ovoj stranici se

nalazi izvornik.

*Zahvaljujem se doc. dr. sc. Marku Čupiću na mentorstvu, pomoći i savjetima pruženim pri pisanju ovog završnog rada.*

# SADRŽAJ

# 1. Uvod

## 1.1. Motivacija

Skoro pa svaka osoba danas posjeduje mobilni uređaj, a neke osobe posjeduju i više njih. Pametni mobilni uređaji postaju neizostavan dodatak svakog modernog čovjeka. Prošlo je vrijeme kada su mobilni uređaji jedino služili za pozive i SMS poruke. Današnji mobilni uređaji imaju puno više mogućnosti. Osim što je zadržana funkcionalnost uspostavljanja poziva i slanja poruka, postoji mogućnost povezivanja na internet, orijentaciju u prostoru, mjerenje brzine itd. Da bi sve te stvari bile moguće, većina današnjih mobilnih uređaja u sebi sadrži senzore koji su navedeni i opisani u tablici 1.1.

Pretpostavimo da je mobilni uređaj spojen na internet i da svakih nekoliko sekundi pošalje vrijednost koju u tom trenutku mjeri pojedini senzor. U samo jednom danu može se skupiti dosta podataka. A što kada to ne bi radili za jedan uređaj nego za sve izdane uređaje nekog modela? Količina podataka bi jako brzo narasla.

Kako količina podataka postaje sve veća, dolazimo do pojma *Velika količina podataka* (engl. *Big Data*). U današnje vrijeme postoji više podataka u digitalnom obliku nego što ih je ikada bilo. Jedan od zanimljivijih izazova je kako ih djelotvorno obraditi i zaključiti nešto iz toga, odnosno kako od te velike količine podataka doći do nekih pametnih zaključaka i nešto novo naučiti.

Tehnologija *Apache Spark* je tehnologija otvorenog koda (engl. *open source*) koja omogućava pisanje programa za obradu podataka u tri programskih jezika: Java, Python i Scala. Dodatno, postoji i mogućnost interaktivnog rada.
U okviru ovog rada proučeni su neki djelovi ove tehnologije, razrađeno nekoliko konkretnih primjera obrade podataka te ostvarena programska rješenja koja obavljaju tu obradu koristeći tehnologiju *Apache Spark*.

Svi primjeri su napisani u programskom jeziku Java.

**Tablica 1.1:** Neki od senzora u današnjim mobilnim uređajima.

| Senzor | Opis |
|---|---|
| Akcelerometar | Elektromehanička komponenta koja mjeri sile ubrzanja. |
| Barometar | Mehanički senzor za mjerenje atmosferskog pritiska (na trenutnoj lokaciji uređaja). |
| Senzor svjetlosti | Mjeri intenzitet, tj. jačinu svjetlosti i uglavnom se nalazi s prednje strane uređaja, iznad ekrana. |
| Senzor blizine | U stanju prepoznati situacije kada mu neki objekt stoji u blizini - ovo omogućava automatske pozive prilikom primicanja telefona licu uz zaključavanje telefona da bi onemogućili slučajno prekidanje poziva uhom ili slično. |
| Senzor gestikulacije | Prepoznaje kretnje ruke tako što detektira infracrvene zrake koje se reflektiraju, odnosno omogućuje djelomično upravljanje telefonom bez doticanja ekrana. |
| Žiroskop | Uređaj koji se koristi za navigaciju i mjerenje kutne brzine. |
| Geomagnetski senzor | Mjeri okolno geomagnetsko polje za sve tri fizičke osi, odnosno služi kao kompas na mobilnim uređajima. |
| *Hall Sensor* | Magnetski senzor zadužen za prepoznavanje je li maska telefona zatvorena ili otvorena. |

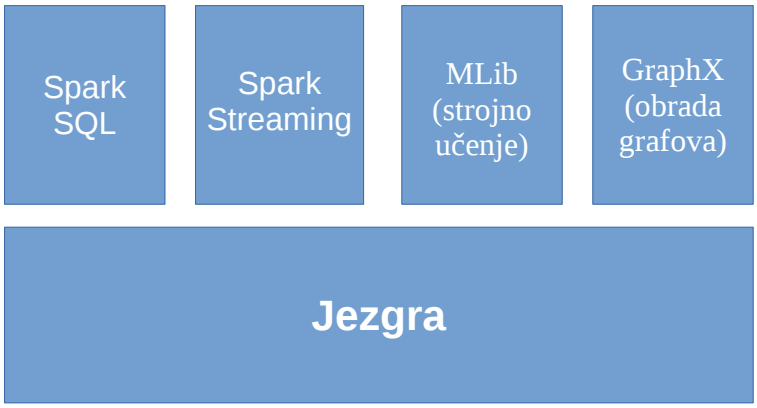
## 1.2. Osnovni gradivni elementi

Tehnologija *Apache Spark* je pisana u programskom jeziku Scala i izvršava se na *Javinom virtualnom stroju* (engl. *Java Virtual Machine*, kratica JVM). Instalacija na osobno računalo je objašnjena u dodatku A. Opisi nekih direktorija i datoteka koje se dobiju instalacijom dani su u tablici 1.2. Zanimljivo je spomenuti da postoji interaktivna ljuska *Spark shell*, ali isključivo za programske jezike Python i Scala. Datoteke u direktoriju *bin* služe upravo za to. Budući da je ovaj rad ograničen isključivo na programski jezik Java, a u ovom trenutku takva interaktivna ljuska još ne postoji, interaktivna ljuska nije detaljnije obrađena. Više informacija o ljusci potražiti u [2].

**Tablica 1.2:** Dio datoteka i direktorija dobivenih instalacijom.

| Datoteka ili direktorij | Opis |
|---|---|
| *bin* | Sadrži izvršive datoteke koje se koriste za interaktivni rad s tehnologijom *Apache Spark*. |
| *core, streaming, python, ...* | Sadrži glavne komponente tehnologije. |
| *README.md* | Sadrži kratke instrukcije za upoznavanje s tehnologijom. |
| *examples* | Sastoji se od nekoliko jednostavnih primjera koje pomažu korisniku da se uhoda i što bezbolnije nauči koristiti programsko sučelje koje tehnologija pruža. |

Osnovna programska apstrakcija s kojom tehnologija *Apache Spark* radi je *otporni raspodijeljeni skup podataka* (engl. *resilient distributed dataset*, kratica RDD) [4]. Atribut *otproni* znači da se može ponovno rekonstruirati u slučaju da se particija uništi. *Raspodijeljeni* znači da je to skup podataka koji se nalazi na računalima (jednom ili više njih) i moguće ga je paralelno obrađivati. *Skup podataka* znači da predstavlja nekakvu kolekciju podataka. Tehnologija *Apache Spark* nudi bogato programsko sučelje za rad s tim skupovima podataka.




**Slika 1.1:** Osnovni elementi tehnologije *Apache Spark*.

Tehnologija se sastoji od nekoliko ključnih elemenata koji su u tablici 1.3 samo nabrojani i opisani rečenicom ili dvije. Detaljnije objašnjenje nalazi se u službenoj dokumentaciji [1]. Slika 1.1 predstavlja osnovne elemente tehnologije *Apache Spark*.

**Tablica 1.3:** Dio datoteka i direktorija dobivenih instalacijom.

| Komponenta | Opis |
|---|---|
| *Spark SQL* | Omogućuje rad s bilo kakvim strukturiranim podatcima, primjerice *JSON*. Također, nudi i mogućnost izvršavanja *SQL* naredbi. |
| *Spark Streaming* | Komponenta zadužena za rad s tokovima podataka. |
| *MLib* | Koristi se za postupke strojnog učenja. |
| *GraphX* | Biblioteka za obradu grafova (npr. graf prijatelja na društvenoj mreži). |

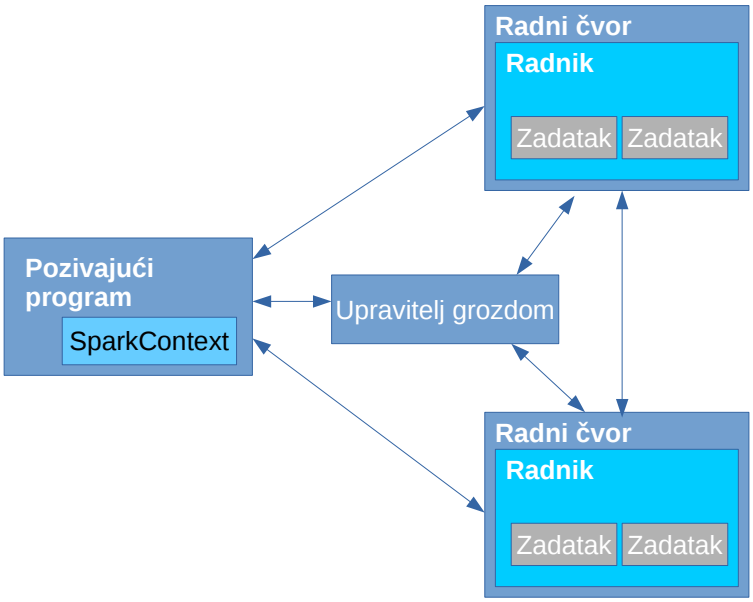
# 2. Prvi programi

U ovom poglavlju opisano je kako napisati osnovni program koristeći tehnologiju *Apache Spark*. Također je opisano i od čega se takav program sastoji te koja je uloga pojedine komponente programa. Primjer koji se obrađuje je primjer 2.1, a svodi se na jednostavan algoritam brojanja riječi.

## 2.1. Postavljanje temelja

### 2.1.1. Osnovni elementi aplikacije

Općenito govoreći, svaka se Spark aplikacija sastoji od nekoliko komponenata. Prva komponenta koju ćemo spomenuti je program koji se izvršava - onaj čija je `main` metoda pokrenuta, odnosno onaj koji pokreće obradu podataka. Taj program naziva se pozivajući program (engl. *driver*). Pozivajući program vrši obradu podataka kroz *SparkContext*.




**Slika 2.1:** Prikaz elemenata aplikacije.

Budući da je tehnologija *Apache Spark* namijenjena za paralelnu obradu podataka, postoje još dvije komponente koje pridonose upravo tome. Pojedino računalo u *grozdu* (engl. *cluster*) naziva se *radnim čvorom* (engl. *worker node*), a proces koji se izvršava na pojedinom računalu naziva se *radnik* (engl. *executor*). Dozvoljeno je da *radnici* međusobno komuniciraju. U cijeloj priči može, a i ne mora eksplicitno biti uključen *upravitelj grozdom* (engl. *cluster manager*). Opisana struktura prikazana je na slici 2.1.

### 2.1.2. Korištenje tehnologije Apache Spark kroz programski jezik Java

Kako bi mogli pisati Spark programe u programskom jeziku Java, potrebno je imati biblioteke koje nisu sastavni dio platforme JDK. Jedna mogućnost je potražiti ih na internetu te ih ručno dohvatiti, raspakirati i uvesti u projekt. Druga mogućnost je koristiti automatizaciju razvojnog ciklusa. Jedan od alata koji to omogućuje je Maven. Za postavljanje i instalaciju Mavena konzultirati poveznicu [1] i knjigu [3]. Za vrijeme pisanja ovog rada, najnovija verzija Sparka je 1.6.1, a odgovarajuće Maven koordinate su:

```
groupId = org.apache.spark
artifactId = spark-core_2.10
version = 1.6.1
```

Odgovarajuća `pom.xml` datoteka nalazi se u dodatku B. Jednom kada je `pom.xml` datoteka namještena i projekt uspješno povezan sa `spark-core`, sve što je još potrebno jest inicijalizirati *SparkContext* i napisati prvu aplikaciju. U primjeru 2.1 nalazi se jednostavna aplikacija koja jedino što radi je broji koliko puta se pojedina riječ pojavljuje u tekstualnoj datoteci.

```
1 package hr.fer.zemris;
2
3 import java.util.Arrays;
4
5 import org.apache.spark.SparkConf;
6 import org.apache.spark.api.java.*;
7 import org.apache.spark.api.java.function.*;
8
```

[1] https://maven.apache.org/install.html

```
9  import scala.Tuple2;
10
11 /**
12  * Razred ima svrhu prikazati osnovnu funkcionalnost Spark
13  * tehnologije na primjeru prebrojavanja riječi
14  * u tekstualnoj datoteci. Kao rezultat program će
15  * zapisati u tekstualnu datoteku koja riječ se koliko puta
16  * ponavlja. Očekuje se dva argumenta kroz naredbeni
17  * redak, a to su putanja do tekstualne datoteke u kojoj
18  * treba izbrojati riječi i putanja do direktorija
19  * u koji će se zapisati rezultat.
20  *
21  * @author mmatak
22  *
23  */
24 public class BrojanjeRijeci {
25   /**
26    * Metoda koja se pokrene kada se pokrene program.
27    * Očekuje putanju do datoteke s riječima i putanju
28    * do direktorija gdje će zapisati rezultat
29    * izvođenja programa.
30    *
31    * @param args
32    *         Argumenti naredbenog retka.
33    */
34   @SuppressWarnings("serial")
35   public static void main(String[] args) {
36     if (args.length != 2) {
37       System.out.println("Program očekuje 2 argumenta.");
38     }
39     String ulaznaDatoteka = args[0];
40     String izlazniDirektorij = args[1];
41
42     // inicijalizacija SparkContext-a
43     SparkConf conf = new SparkConf()
44               .setMaster("local")
45               .setAppName("Brojanje rijeci");
46     JavaSparkContext sc = new JavaSparkContext(conf);
```

```
47
48      // učitavanje podataka
49      JavaRDD<String> ulaz = sc.textFile(ulaznaDatoteka);
50
51      // razmak se koristi da bi razdvojio dvije riječi
52      JavaRDD<String> rijeci = ulaz.flatMap(
53        new FlatMapFunction<String, String>() {
54          public Iterable<String> call(String redak) {
55            return Arrays.asList(redak.split(" "));
56          }
57        }
58      );
59
60      // transformiraj u parove (rijec,1) i broji
61      JavaPairRDD<String, Integer> brojRijeci = rijeci
62      .mapToPair(new PairFunction<String, String, Integer>() {
63        public Tuple2<String, Integer> call(String rijec) {
64          return new Tuple2<String, Integer>(rijec, 1);
65        }
66      }).reduceByKey(
67        new Function2<Integer, Integer, Integer>() {
68          public Integer call(Integer x, Integer y) {
69            return x + y;
70          }
71        }
72      );
73
74      // spremi rezultat u izlaznu datoteku
75      brojRijeci.saveAsTextFile(izlazniDirektorij);
76      // zatvori SparkContext
77      sc.close();
78    }
79  }
```

**Primjer 2.1:** Program koji broji koliko se puta koja riječ pojavljuje u datoteci.

Analizirajmo što smo napravili. Inicijalizirali smo *SparkContext* tako što smo rekli da se odvija na lokalnom računalu i zadali smo ime aplikacije. Zatim smo kreirali

RDD iz tekstualne datoteke koja je predana kao argument naredbenog retka. Taj skup podataka nam je poslužio za kreiranje novog skupa podataka koji je nastao tako što smo svaki redak razdvojili po razmaku i kreirali uređeni par (*riječ*, 1). U tom uređenom paru, koji je oblika (*ključ*, *vrijednost*), *riječ* nam predstavlja ključ, a 1 predstavlja vrijednost. Zatim smo iz tako uređenih parova, one parove koji imaju jednaki ključ zbrojili po vrijednostima i u tom trenutku[2] nije više postojalo dva ili više uređena para koja imaju jednake ključeve. Na kraju smo rezultat zapisali i eksplicitno zatvorili *SparkContext*.

Kako je ovo bio početni primjer, *lambda* izrazi koji su uobičajeni za inačicu 8 programskog jezika *Java* nisu korišteni iz razloga da bi se lakše shvatilo što se sve treba implementirati. Kod iz primjera 2.1 od retka 51 do retka 72 koristeći lambda izraze dan je u primjeru 2.2.

```
// razmak se koristi da bi razdvojio dvije riječi
JavaRDD<String> rijeci = ulaz
          .flatMap(
            redak -> Arrays.asList(" ")
          );

// transformiraj u parove (rijec,1) i broji
JavaPairRDD<String, Integer> brojRijeci = rijeci
          .mapToPair(
            rijec -> new Tuple2<String, Integer>(rijec, 1)
          ).reduceByKey(
             (x, y) -> x + y
          );
```

**Primjer 2.2:** Brojanje riječi koristeći lambda izraze.

## 2.2. Otporni raspodijeljeni skup podataka

*Otporni raspodijeljeni skup podataka* (engl. *resilient distributed dataset*, kratica RDD) je osnovna podatkovna struktura tehnologije Apache Spark. To je *nepromjenjiva* (engl. *im-*

[2]U tom trenutku se ništa dogodilo nego tek nakon poziva metode `saveAsTextFile()`, ali *lijena evaluacija* (engl. *lazy evaluation*) opisana je tek kasnije.

*mutable, read-only*) kolekcija podataka. Iz toga proizlazi da se iz jednog skupa podataka može jedino napraviti drugi skup podataka, a ne može se promijeniti postojeći.

U prethodnom primjeru to je vidljivo pri pozivu metode `flatMap`. Ta metoda transformira `ulaz` u `rijeci`. Sličnu transformaciju radi i metoda `mapToPair`. Drugim riječima, *transformacija* (engl. *transformation*) je svaka metoda koja iz jednog skupa podataka kreira novi skup. Uz transformacije, postoje i *akcije* (engl. *actions*). Akcije se razlikuju od transformacija po tome što ne vraćaju novi skup podataka nego prvenstveno služe za dohvat jednog ili više elemenata iz nekog skupa. Dodatno, imaju mogućnost zapisivanja podataka kao što je prikazano u primjeru 2.1 - metoda `saveAsTextFile`.

U tablici 2.1 mogu se naći neke od mogućih transformacija i njihovi opisi, a u tablici 2.2 nalaze se akcije koje je moguće pozvati zajedno s odgovarajućim opisima.

**Tablica 2.1:** Neke od transformacija nad otpornim raspodijeljenim skupovima podataka.

| Transformacija | Opis |
|---|---|
| `map(`*`func`*`)` | Vraća novi skup nastao tako što je svaki element orginalnog skupa predan funkciji *func*. |
| `filter(`*`func`*`)` | Vraća novi skup nastao tako što su iz orginalnog skupa preuzeti samo oni elementi za koje funkcija *func* vraća `true`. |
| `flatMap(`*`func`*`)` | Slično kao `map(`*`func`*`)`, ali jedan ulazni element kreira 0 ili više izlaznih elemenata. |
| `union(`*`drugiSkup`*`)` | Vraća uniju između skupa nad kojim je transformacija pozvana i skupa koji je predan kao argument. |
| `intersection(`*`drugiSkup`*`)` | Vraća presjek između skupa nad kojim je transformacija pozvana i drugog skupa koji je predan kao argument. |

Detaljnije objašnjenje ovih i ostalih transformacija potražiti na poveznici[3].

Zbog velike količine podataka, evaluacija transformacija je *lijena* (engl. *lazy evaluation*). Lijena evaluacija potječe iz funkcijskih jezika. O korisnosti lijene evaluacije kao programskog alata raspravljeno je u [Friedman i Wise, 1976; Henderson i Morris,

[3] `http://spark.apache.org/docs/latest/programming-guide.html#transformations`

1976; Henderson, 1980; Hudak, 1989; Bird i Wadler, 1988; Field i Harrison, 1988; O'Donnell, 1985]. Lijena evaluacija je evaluacija *na zahtjev* - obavlja se računanje samo onda i onoliko koliko je potrebno da se dobije traženi izlaz. Programer može definirati ogromne strukture podataka kao međurezultate i računati na implementaciju jezika u kojem programira da će se zapravo računati samo potrebni dijelovi tih struktura podataka. To znači da Spark samo pamti koje sve transformacije treba napraviti nad skupovima, ali ne i da ih odmah odradi. Sve potrebne transformacije rade se na zahtjev, odnosno tek pozivom prve akcije.

**Tablica 2.2:** Akcije primjenjive nad otpornim raspodijeljenim skupom podataka

| Akcija | Opis |
|---|---|
| `reduce(`*`func`*`)` | Agregacija elemenata iz skupa tako što se nad dva elementa pozove funkcije *func*, a ta funkcija vrati jedan element. Funkcija *func* bi trebala biti komutativna i asocijativna kako bi se pravilno izvršavala u paralelnoj obradi podataka. |
| `collect()` | Vraća polje svih elemenata iz skupa direktno u pozivajući program. Budući da je tih elemenata puno, u praksi se poziva nakon transformacije `filter()`. |
| `count()` | Vraća broj elemenata u skupu. |
| `take(`*`n`*`)` | Vraća polje prvih *n* elemenata iz skupa. |
| `first()` | Vraća prvi element iz skupa. Može se ostvariti i pozivom akcije `take(`*`1`*`)` |

Detaljnije objašnjenje ovih i ostalih akcija potražiti na poveznici[4].

Kao primjer lijene evaluacije, može poslužiti jedna od čestih implementacija oblikovnog obrasca *jedinstveni objekt* (engl. *singleton*). Takva implementacija prikazana je u primjeru 2.3. Ono što je ovdje *lijeno* je kreiranje instance razreda `MySingleton` - kreirana je tek na zahtjev tj. prilikom prvog poziva metode `getInstance()`.

[4] `http://spark.apache.org/docs/latest/programming-guide.html#actions`

```
1  package hr.fer.zemris.prviProgrami;
2
3  public final class MySingleton {
4    private static MySingleton instanca;
5
6    private MySingleton() {
7      // privatni konstruktor
8    }
9
10   public static MySingleton getInstance() {
11     if (instanca == null) {
12       instanca = new MySingleton();
13     }
14     return instanca;
15   }
16 }
```

**Primjer 2.3:** Lijena evaluacija u oblikovnom obrascu *jedinstveni objekt*.

Treba uzeti u obzir da ovakva implementacija oblikovnog obrasca *jedinstveni objekt* **nije sigurna** za višedretveno okruženje. Može se dogoditi scenarij gdje jedna dretva taman završi s provjerom na liniji 11 i kao rezultat dobije `true` tj. krene s izvršavanjem linije 12. Prije nego je linija 12 izvršena, druga dretva uspješno prođe kroz provjeru u retku 11 i isto tako krene izvršavati redak 12. Posljedica je da su kreirane dvije instance razreda, a to se krši s osnovnim načelom ovog oblikovnog obrasca - smije postojati samo jedna instanca razreda.

U nastavku slijedi primjer 2.5 koji obrađuje datoteku `logfile.txt`. Jedan od redaka te datoteke dan je u primjeru 2.4 i iz toga se može zaključiti kako općenito izgleda redak datoteke `logfile.txt`.

```
89.164.244.106 – [24/Feb/2008:01:05:08] "GET /index.jsp?lang=hr HTTP/1.1" 200
```

**Primjer 2.4:** Korištenje transformacija i akcija.

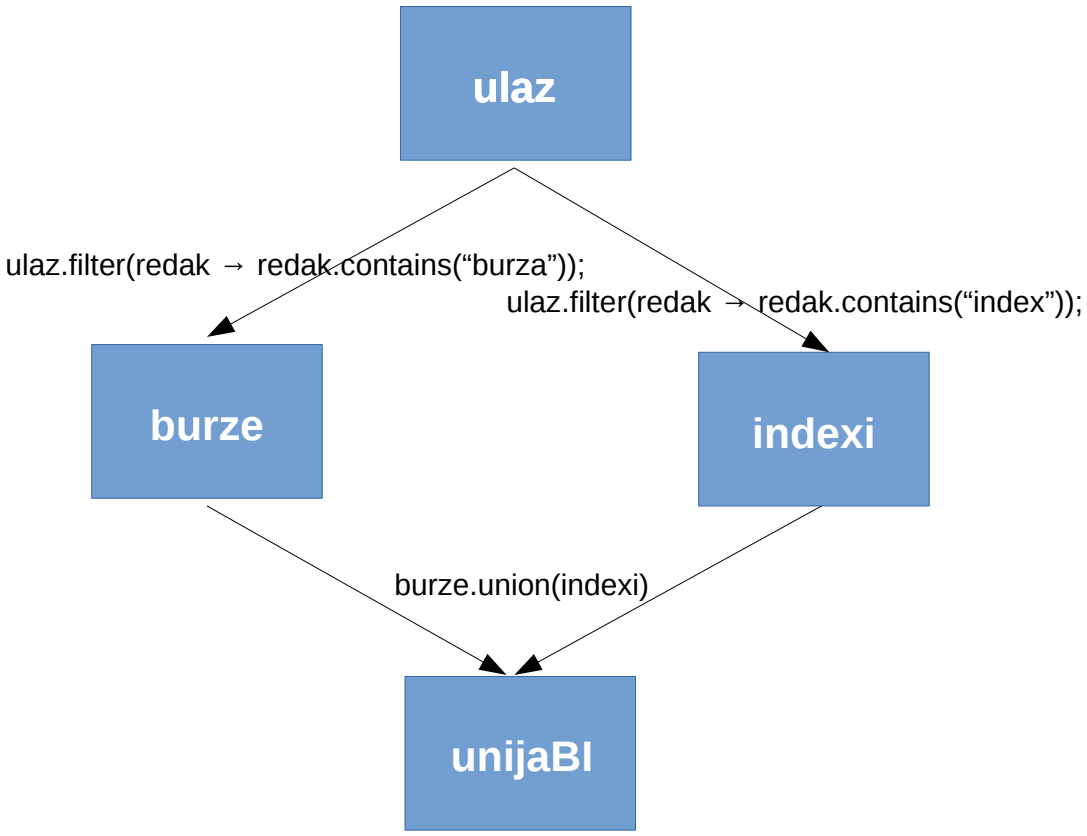
Primjer 2.5 broji koliko puta je zahtjev bio na URL koji u sebi sadrži riječ "burza" i koliko je puta došao zahtjev na URL koji u sebi sadrži riječ "index". Njihova unija je također izračunata.

```java
package hr.fer.zemris;

import org.apache.spark.SparkConf;
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.JavaSparkContext;

public class Primjer3 {
  public static void main(String[] args) {
    // inicijalizacija SparkContext-a
    SparkConf conf = new SparkConf()
        .setMaster("local")
        .setAppName("Brojanje rijeci");
    JavaSparkContext sc = new JavaSparkContext(conf);

    // učitavanje podataka
    JavaRDD<String> ulaz = sc.textFile("logfile.txt");

    // transformacije
    JavaRDD<String> burze = ulaz.filter(
        redak -> redak.contains("burza")
      );
    JavaRDD<String> indexi = ulaz.filter(
        redak -> redak.contains("index")
      );
    JavaRDD<String> unijaBI = burze.union(indexi);

    // akcije
    long brojLinija = unijaBI.count();
    long ukupanBrojLinija = ulaz.count();
    System.out.printf(
        "Broj linija koje sadrže riječ 'burza' ili 'index'
         je: %d, odnosno %f%%.\n", brojLinija,
         (double) 100 * brojLinija / ukupanBrojLinija
         );
    System.out.printf(
        "Prva linija koja sadrži riječ 'burza' ili 'index'
        je: %s\n", unijaBI.first()
```

```
            );
        sc.close();
    }
}
```

**Primjer 2.5:** Korištenje transformacija i akcija.

Ovdje imamo 4 skupa podataka: `ulaz`, `burze`, `indexi` i `unijaBI`. Skupovi podataka `burze` i `indexi` su kreirani na temelju skupa podataka `ulaz`, a `unijaBI` je kreiran na temelju `burze` i na temelju `indexi`. Na slici 2.2 nalazi se graf koji to opisuje.




**Slika 2.2:** Transformacije nad skupovima podataka.

Tek prilikom poziva akcije `unijaBI.count()` se zapravo kreira `ulaz`, na temelju njega `burza` i `indexi` i onda tek na temelju njih se kreira `unijaBI`. Iz `unijaBI` ukupni broj linija dohvati se pozivom `unijaBI.count()`. Nakon što se to izračuna, svi skupovi podataka nestaju iz memorije. Prilikom poziva akcije `unijaBI.first()` **ponovno** kreće obrada podataka gore navedenim redosljedom i sve ide iz početka.

Na prvi pogled ovo izgleda kao loša implementacija, ali budući da se ovdje radi o velikoj količini podataka i ne možemo ih nikako sve pohraniti u memoriju, ovo je zapravo logična implementacija. Ukoliko ne želimo svaki puta iz početka računati i kreirati `unijaBI`, trebamo ga pohraniti pozivom metode `persist()` i predavanjem odgovarajućeg parametra. Tablica 2.3 sadrži usporedbu različitih parametara koji se mogu predati metodi `persist()`.

**Tablica 2.3:** Usporedba mogućih parametara za metodu `persist()`.

| Razina | Prostorno zauzeće | Procesorsko vrijeme | U memoriji | Na disku |
|---|---|---|---|---|
| `MEMORY_ONLY` | Visoko | Nisko | Da | Ne |
| `MEMORY_ONLY_SER` | Nisko | Visoko | Da | Ne |
| `MEMORY_AND_DISK` | Visoko | Srednje | Dio | Dio |
| `MEMORY_AND_DISK_SER` | Nisko | Visoko | Dio | Dio |
| `DISK_ONLY` | Nisko | Visoko | Ne | Da |

Iz tablice 2.3 očito je da postoje dvije osi usporedbe: spremanje na disk - spremanje u memoriju i spremanje u serijaliziranom obliku - spremanje u neserijaliziranom obliku. Postoji i parametar `MEMORY_AND_DISK` koji prvo podatke sprema u memoriju dok se memorija ne napuni, a kada se napuni, ostatak podataka se sprema na disk. Ono što razlikuje parametre koji sadrže sufiks `SER` i one koji ga ne sadrže je to što parametri sa sufiksom `SER` spremaju podatke u serijaliziranom obliku, a drugi ne.

# 3. Napredno programiranje

U ovom poglavlju opisane su neke napredne tehnike za rad s tehnologijom *Apache Spark*. Objašnjen je algoritam *PageRank*, a dana je i implementacija istog u primjeru 3.1.

## 3.1. Primjer: Algoritam PageRank

Zanimljivo je pitanje kako je *Google* toliko dobar u rezultatima koje korisnik dobije na svoj upit, točnije, kako ispravno sortira po relevantnosti stranice od važnije prema manje važnoj? Odgovor na to daje algoritam *PageRank* koji je dobio ime po jednom od osnivača *Googlea*, Larry Pageu. Ovo potpoglavlje analizira upravo algoritam *PageRank*. U nastavku je iznesena pojednostavljena verzija algoritma, a za više informacija konzultirati [5].

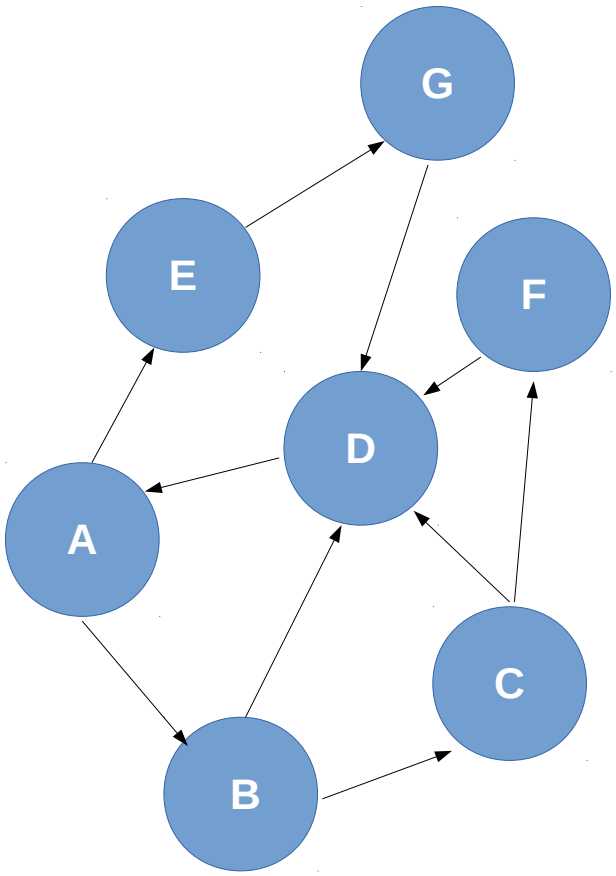
### 3.1.1. Upoznavanje s algoritmom

Algoritam računa koliko je koja stranica važna po tome koliko drugih stranica upućuje na nju. Ideja je jednostavna: što više stranica ima poveznicu na neku stranicu `N`, to je stranica `N` važnija i time bolje rangirana. U obzir se uzima i koja stranica pokazuje na stranicu `N` (je li to stranica na koju sve ostale stranice pokazuju ili je to neka stranica na koju nitko ne pokazuje) kao i na koliko ostalih stranica ta stranica sadrži poveznice (nije isto ako je na cijeloj stranici samo jedna poveznica i ako na cijeloj stranici ima 1000 poveznica) i koliko su one same važne. Stranice na koje neka stranica `M` sadrži poveznice nazivamo *susjedima* te stranice. Pojednostavljena izvedba prikazana je u nastavku.

1. Početni rang svake stranice postavi se na 1.0.
2. Ponavljaj `P` puta:
   2.1. Svaka stranica *n* svim svojim susjedima šalje doprinos `rang(`*n*`) / brojSusjeda(`*n*`)`.
   2.2. Postavi ukupni rang stranice prema formuli: $0.15 + 0.85 * \textit{ukupan primljeni doprinos}$.

Kako bi se što brže dobio što precizniji rezultat, važno je broj ponavljanja algoritma `P` postaviti na prikladnu vrijednost. U praksi je dovoljno `P` postaviti na 10.

Pretpostavimo raspored stranica kao na slici 3.1.




**Slika 3.1:** Raspored susjeda u algoritmu

U nastavku je raspisan algoritam *PageRank* nad stranicama koje imaju susjede definirane kao na slici 3.1. Ako stranica `N` sadrži poveznicu na stranicu `M` to je na toj istoj slici definirano tako da strelica počinje iz kruga s oznakom `N`, a vrh joj pokazuje na krug s oznakom `M`.

Intuitivno se čini da bi stranica D trebala biti najviše rangirana budući da najviše stranica upućuje na nju. Idemo to provjeriti. Za početak, odredimo susjede od svake stranice i postavimo rang svake stranice na 1.0 kao što je prikazano u tablici 3.1.

**Tablica 3.1:** Inicijalizacija rangova i određivanje susjeda

| Stranica | Rang | Susjedi |
|---|---|---|
| A | 1.0 | B, E |
| B | 1.0 | C |
| C | 1.0 | D, F |
| D | 1.0 | A |
| E | 1.0 | G |
| F | 1.0 | D |
| G | 1.0 | D |

Sada se ponavljaju koraci 2.1 i 2.2 P puta. Ovdje je ponovljeno 2 puta i to bi trebalo biti dovoljno za razumjevanje algoritma. Iz koraka 2.1 računa se doprinos koja šalje svaka od stranica. Doprinos stranice A koji ona šalje drugima iznosi $1.0/2 = 0.5$. Doprinos stranice B koji ona šalje drugima je $1.0/2 = 0.5$. Doprinos stranice C jednak je kao i doprinos stranica A i B jer imaju jednak rang i broj susjeda. Doprinos stranica D, E, F i G iznosi $1.0/1 = 1.0$. Rezultati za svaku od stranica prikazani su u tablici 3.2.

**Tablica 3.2:** Izračun doprinosa koji šalje svaka od stranica.

| Stranica | Rang | Susjedi | Doprinos drugima |
|---|---|---|---|
| A | 1.0 | B, E | 0.5 |
| B | 1.0 | C, D | 0.5 |
| C | 1.0 | D, F | 0.5 |
| D | 1.0 | A | 1.0 |
| E | 1.0 | G | 1.0 |
| F | 1.0 | D | 1.0 |
| G | 1.0 | D | 1.0 |

Sada, za svaku od stranica se računa ukupna suma koju je ta stranica dobila od svojih susjeda po formuli danoj u koraku 2.2. Budući da je stranica A susjed samo od stranice D, ukupni primljeni doprinos koji je primila stranica A je upravo doprinos koji stranica D može dati drugima, a to je 1.0. Stranica B je susjed jedino stranici A i doprinos koji prima je upravo doprinos koji stranica A može u ovom trenutku dati

drugima, a to je $0.5$. Stranica `C` je susjed jedino stranici `B` i zbog toga je njen ukupni primljeni doprinos $0.5$. Stranica `D` je susjed stranicama `B`, `C`, `F` i `G` pa je njen ukupni primljeni doprinos $0.5 + 0.5 + 1.0 + 1.0 = 3.0$. Ukupan primljeni doprinos za svaku od stranica nalazi se u tablici 3.3.

**Tablica 3.3:** Ukupni primljeni doprinos svake od stranica.

| Stranica | Rang | Susjedi | Doprinos drugima | Ukupni primljeni doprinos |
|---|---|---|---|---|
| A | 1.0 | B, E | 0.5 | 1.0 |
| B | 1.0 | C, D | 0.5 | 0.5 |
| C | 1.0 | D, F | 0.5 | 0.5 |
| D | 1.0 | A | 1.0 | 3.0 |
| E | 1.0 | G | 1.0 | 0.5 |
| F | 1.0 | D | 1.0 | 0.5 |
| G | 1.0 | D | 1.0 | 1.0 |

Nakon što je izračunat ukupni primljeni doprinos, vrijeme je za ponovni izračun ranga svake od stanica po formuli koja se nalazi u koraku 2.2. Novi rang stranice `A` iznosi $0.15 + 0.85 * 1.0 = 1.0$. Novi rang stanice `B` iznosi $0.15 + 0.85 * 0.5 = 0.575$. Novi rang stranice `C` iznosi isto $0.575$ jer stranica `C` i stranica `B` imaju jednaki ukupni primljeni doprinos. Novi rang stranice `D` je $0.15 + 0.85 * 3.0 = 2.7$. Nakon što se izračuna i postavi novi rang za svaku od stranica, nastupi situacija kao u tablici 3.4

**Tablica 3.4:** Rang svake od stranica nakon prve iteracije.

| Stranica | Rang | Susjedi |
|---|---|---|
| A | 1.0 | B, E |
| B | 0.575 | C, D |
| C | 0.575 | D, F |
| D | 2.7 | A |
| E | 0.575 | G |
| F | 0.575 | D |
| G | 1.0 | D |

Vidimo da je već u prvoj iteraciji algoritma stranica `D` iskočila sa svojim rangom od okoline. Ovo smo bili i pretpostavili budući da je stranica `D` susjed najvećem broju stranica, odnosno najveći broj stranica sadrži poveznicu na stranicu `D`.

Idemo napraviti još jednu iteraciju i vidjeti što će se dogoditi s rangovima.

Iz tablice 3.4 dohvaćamo rang i broj susjeda svake od stranica te računamo doprinos koji svaka od stranica može dati svojim susjedima prema već spomenutoj formuli u gornjem algoritmu u koraku 2.1. Doprinos koji stranica A šalje drugima iznosi $1.0/2 = 0.5$. Doprinos koji stranica B šalje je $0.575/2 = 0.2875$. Doprinos stranice C je jednak doprinosu stranice B budući da imaju jednak rang i broj susjeda. Doprinos stranice D iznosi $2.7/1 = 2.7$. Doprinos stranica E i F iznosi $0.575$, a doprinos stranice G je $1.0$. Rezultati izračuna koliko koja stranica doprinosi drugima prikazani su u tablici 3.5.

**Tablica 3.5:** Izračun doprinosa koji šalje svaka od stranica u drugoj iteraciji.

| Stranica | Rang | Susjedi | Doprinos drugima |
|---|---|---|---|
| A | 1.0 | B, E | 0.5 |
| B | 0.575 | C, D | 0.2875 |
| C | 0.575 | D, F | 0.2875 |
| D | 2.7 | A | 2.7 |
| E | 0.575 | G | 0.575 |
| F | 0.575 | D | 0.575 |
| G | 1.0 | D | 1.0 |

Sada se ponovno računa ukupni primljeni doprinos svake od stranica. Tako za stranicu A se dobije da ukupni primljeni doprinos iznosi $2.7$ jer je ona jedino susjed stranici D. Ukupni primljeni doprinos stranice B je $0.5$ jer je upravo to iznos koliko stranica A doprinosi drugima, a stranica B je jedino susjed od stranice A. Stranica C ima ukupni primljeni doprinos $0.2875$ jer je upravo to iznos doprinosa stranice B, a stranica C je susjed jedino stranici B. Stranica D prima zbroj doprinosa svih stranica kojima je susjed odnosno zbroj doprinosa stranica B, C, F i G. Iz toga proizlazi da je ukupni primljeni doprinos stranice D $0.2875 + 0.2875 + 0.575 + 1.0 = 2.15$. Ukupni primljeni doprinos za svaku od stranica nalazi se u tablici 3.6.

**Tablica 3.6:** Ukupni primljeni doprinos svake od stranica u drugoj iteraciji.

| Stranica | Rang | Susjedi | Doprinos drugima | Ukupni primljeni doprinos |
|---|---|---|---|---|
| A | 1.0 | B, E | 0.5 | 2.7 |
| B | 0.575 | C, D | 0.2875 | 0.5 |
| C | 0.575 | D, F | 0.2875 | 0.2875 |
| D | 2.7 | A | 2.7 | 2.15 |
| E | 0.575 | G | 0.575 | 0.5 |
| F | 0.575 | D | 0.575 | 0.2875 |
| G | 1.0 | D | 1.0 | 0.575 |

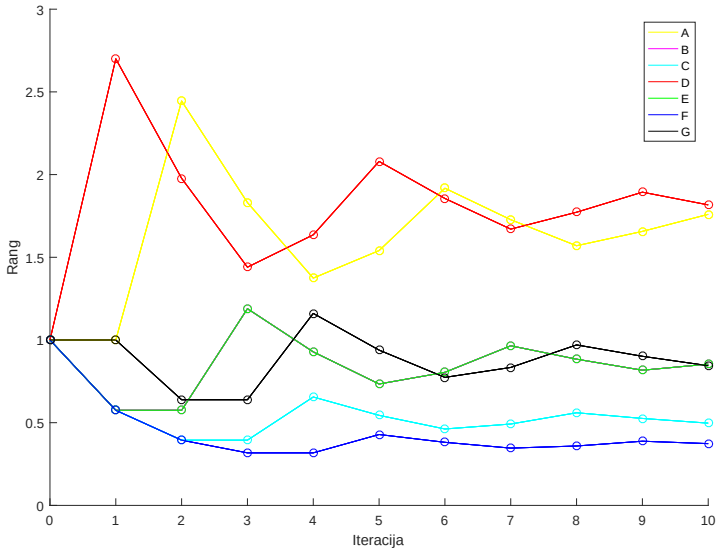
Nakon što je i u drugoj iteraciji izračunat ukupni primljeni doprinos, vrijeme je za ponovni izračun ranga svake od stanica po formuli koja se nalazi u koraku 2.2. Novi rang stranice `A` iznosi $0.15 + 0.85 * 2.7 = 2.445$. Novi rang stanice `B` iznosi $0.15 + 0.85 * 0.5 = 0.575$. Novi rang stranice `C` iznosi $0.15 + 0.85 * 0.2875 = 0.394375$. Novi rang stranice `D` je $0.15 + 0.85 * 2.15 = 1.9775$. Rang stranice `E` je $0.15 + 0.85 * 0.5 = 0.575$, a rang stranice `F` iznosi $0.15 + 0.85 * 0.2875 = 0.394375$. Konačno, rang stranice `G` je $0.15 + 0.85 * 0.575 = 0.63875$. Nakon što se postavi novi rang za svaku od stranica, nastupi situacija kao u tablici 3.7

**Tablica 3.7:** Rang svake od stranica nakon druge iteracije.

| Stranica | Rang | Susjedi |
|---|---|---|
| A | 2.445 | B, E |
| B | 0.575 | C, D |
| C | 0.394375 | D, F |
| D | 1.9775 | A |
| E | 0.575 | G |
| F | 0.394375 | D |
| G | 0.63875 | D |

Iz tablice 3.7 vidljivo je da je stranica `A` preuzela vodstvo. To se isto može intuitivno prihvatiti budući da skoro sve stranice pokazuju na stranicu `D`, a upravo `D` je jedina stranica koja pokazuje na `A`.

Nakon što se izvrti 10 iteracija, stvari se stabiliziraju. Graf koji opisuje kako se rangovi mijenjaju kroz 10 iteracija prikazan je slikom 3.2.




**Slika 3.2:** Vrijednosti ranga ovisno o iteraciji

### 3.1.2. Implementacija i analiza

Kada je poznato kako "ručno" radi algoritam, ne bi trebao biti problem napisati programsku implementaciju koja se nalazi u primjeru 3.1. U ovom dijelu je osim samog koda napisana i detaljna analiza tog koda kako bi se dobilo što više informacija o tome kako radi tehnologija *Apache Spark*.

```
1  package hr.fer.zemris.naprednoProgramiranje;
2
3  import java.util.ArrayList;
4  import java.util.List;
5
6  import org.apache.spark.SparkConf;
7  import org.apache.spark.api.java.JavaPairRDD;
8  import org.apache.spark.api.java.JavaRDD;
9  import org.apache.spark.api.java.JavaSparkContext;
10
11 import com.google.common.collect.Iterables;
12
13 import scala.Tuple2;
14
15 /**
16  * Program računa rang pojedine stranice.
```

```
17  * Očekuje se da svaki redak ulazne datoteke sadrži
18  * samo 2 stranice u formatu "StranicaA StranicaB", a taj
19  * zapis predstavlja da je StranicaB susjed od StranicaA.
20  *
21  * @author mmatak
22  *
23  */
24 public class AlgoritamPageRank {
25   private static final String ULAZNA_DATOTEKA =
        "pageRankInput.txt";
26   private static final int BROJ_ITERACIJA = 10;
27
28   public static void main(String[] args) {
29     // Inicijalizacija SparkContext-a.
30     SparkConf conf = new SparkConf()
31         .setMaster("local")
32         .setAppName("Algoritam PageRank");
33     JavaSparkContext sc = new JavaSparkContext(conf);
34
35     // Učitavanje podataka.
36     JavaRDD<String> ulaz = sc.textFile(ULAZNA_DATOTEKA);
37
38     // Pročitaj sve ulazne URL-e i inicijaliziraj njihove
          susjede.
39     JavaPairRDD<String, Iterable<String>> linkovi =
          ulaz.mapToPair(
40         redak -> {
41           String[] elementi = redak.split("\\s+");
42           return new Tuple2<String, String>(
43             elementi[0],
44             elementi[1]
45             );
46         }
47       ).distinct().groupByKey().cache();
48     // Svako ime susjeda zamijeni s 1.0 i vrati novi skup
          podataka.
49     JavaPairRDD<String, Double> rangovi =
          linkovi.mapValues(value -> 1.0);
```

```
50
51      // iteracija 2. i 3. koraka algoritma
52      for (int i = 0; i < BROJ_ITERACIJA; i++) {
53        // za svaki link izracunaj njegovu doprinos drugim
             linkovima
54        JavaPairRDD<String, Double> doprinosi =
55        linkovi.join(rangovi).values().flatMapToPair(
56          linkoviRang -> {
57
58            int brojSusjeda = Iterables.size(linkoviRang._1);
59            List<Tuple2<String, Double>> rezultati =
60            new ArrayList<Tuple2<String, Double>>();
61
62              for (String susjed : linkoviRang._1) {
63                rezultati.add(new Tuple2<String, Double>(
64                  susjed,
65                  linkoviRang._2 / brojSusjeda
66                  )
67                );
68              }
69
70              return rezultati;
71          }
72        );
73        rangovi = doprinosi
74            .reduceByKey((v1, v2) -> v1 + v2)
75            .mapValues(suma -> 0.15 + suma * 0.85);
76      }
77      // spremi u memoriju
78      List<Tuple2<String, Double>> rangoviFinal =
           rangovi.collect();
79      // ispis
80      for (Tuple2<String, Double> entry : rangoviFinal) {
81        System.out.printf(
82          "Stranica %s ima rang %.2f\n",
83           entry._1,
84           entry._2
85        );
```

```
86        }
87        sc.close();
88    }
89 }
```

**Primjer 3.1:** Algoritam *PageRank*.

Analizirajmo kod naveden u primjeru 3.1.

Od linije 30 do linije 33 inicijalizira se glavno računalo i ime aplikacije. Ovdje je glavno računalo postavljeno na `local` budući da se aplikacija odvija na lokalnom računalu, a ne na grozdu. Ime aplikacije postavljeno je na temelju algoritma kojeg aplikacija implementira.

Linija 36 obavlja inicijalizaciju *otpornog raspodijeljenog skupa podataka* čitajući podatke iz datoteke čija je putanja zadana varijablom `ULAZNA_DATOTEKA`.

Od linije 39 do linije 47 obavlja se pridruživanje susjeda svakoj stranici. Svaki se redak iz skupa `ulaz` podijeli na dva elementa - element prije i element poslije razmaka. Ta dva elementa predstavljaju uređeni par (*ključ*, *vrijednost*) gdje je stranica koja je prije razmaka *ključ*, a stranica nakon razmaka *vrijednost*. To znači da je stranica *vrijednost* susjed od stranice *ključ*. Budući da su dozvoljeni duplikati u ulaznom skupu podataka tj. može postojati dva identična retka koja govore kako stranica $N$ ima susjeda $M$, to se mora uzeti u obzir. Za rangiranje stranice nije bitno koliko poveznica stranica $N$ sadrži na stranicu $M$. U ovoj implementaciji, svejedno je radi li se o 1 ili 1000 poveznica jer su to sve iste poveznice. Da bi se osiguralo da ne postoji niti jedan duplikat, poziva se metoda `distinct()`.

Pozivom metode `groupByKey()` obavlja se *spajanje* uređenih parova s istim ključem, a njihove vrijednosti spremamo u listu. Primjerice, postoje li kolekcija uređenih parova ($A$, $B$), ($A$, $C$), ($A$, $D$) i nad tom kolekcijom pozove se `groupByKey()` rezultat će biti novi skup podataka predstavljen uređenim parom ($A$, $\{B, C, D\}$). *Vrijednost* tog uređenog para je iterabilna (implementira sučelje `Iterable`) i zbog toga je povratna vrijednost u obliku `JavaPairRDD<String, Iterable<String>>`. Rezultat je skup podataka `linkovi` predstavljen uređenim parovima gdje su *ključevi* stranice, a *vrijednosti* susjedi tih stranica. Konačno, kako se ne bi ovo moralo svaki puta ispočetka računati, rezultat se pohranjuje u memoriju pozivom metode `cache()`.

Linija 49 iz skupa podataka `linkovi` "provuče" svaku od *vrijednosti* (čitava kolekcija je zapravo jedna *vrijednost*) kroz funkciju koja tu čitavu kolekciju samo preslika u *vrijednost* 1.0. Sada je nastao novi skup uređenih parova koji za *ključ* imaju

stranicu, a za *vrijednost* 1.0. Ovim postupkom je zapravo postavljen rang svake stranice na početnu vrijednost.

Poziv naredbe `linkovi.join(rangovi)` u liniji 55 vraća novi skup podataka koji kao *ključeve* ima *ključeve* iz skupova `linkovi` i `rangovi` (bez dupliciranja), a odgovarajuće *vrijednosti* su uređeni parovi *vrijednosti* iz tih skupova. Preciznije rečeno, `join(drugiSkupPodataka)` ako je pozvan nad skupovima uređenih parova oblika (*K*,*V*) i (*K*, *W*) kao rezultat vraća skup podataka uređenih parova oblika (*K*,(*V*, *W*)). Ukoliko postoje različiti *ključevi* u skupovima podataka nad kojima se izvršava `join()`, parovi s stim *ključem* su ignorirani. Jedino se "uparuju" uređeni parovi s istim *ključem*. Metoda `values()` vraća skup svih *vrijednosti* iz skupa podataka dobivenog pozivom metode `join()`. U ovom primjeru metoda `values()` vraća skup uređenih parova gdje je *ključ* kolekcija susjeda neke stranice, a *vrijednost* rang te stranice. Metoda `flatMapToPair()` kao argument prima funkciju koja implementira sučelje `PairFlatMapFunction<T,K2,V2>`. Ta funkcija prima element tipa `T` i vraća nula ili više uređenih parova oblika (`K2`, `V2`). Metoda `flatMapToPair()` zapravo "provuče" svaki element skupa nad kojim je pozvana kroz funkciju koju primi kao argument i na kraju, na temelju uređenih parova koje je ta funkcija vratila, kreira skup podataka oblika (`K2`, `V2`). Element koji se "provlači" imenovan je varijablom `linkoviRang`.

Od linije 56 do linije 70 obrađuju se podatci koji su predstavljeni uređenim parovima oblika (*ključ*, *vrijednost*) gdje je *ključ* kolekcija susjeda neke stranice, a *vrijednost* rang te stranice.

Linija 58 zapravo dohvaća veličinu kolekcije koja je predstavljena *ključem*, odnosno varijablu `brojSusjeda` postavlja na onu vrijednost koliko stranica koja se trenutno obrađuje ima susjeda.

Od linije 62 do linije 68 proteže se petlja koja u listu `rezultati` sprema uređene parove oblika (*ključ*, *vrijednost*) gdje je *ključ* susjed, a *vrijednost* doprinos stranice koja se trenutno obrađuje tom susjedu.

Funkcija `flatMapToPair()` spaja sve vraćene rezultate u jedan skup i sada se u skupu podataka `doprinosi` nalaze uređeni parovi gdje je *ključ* stranica, a *vrijednost* doprinos neke stranice toj stranici. Budući da neka stranica ne mora biti susjed samo jednoj stranici, u tom skupu podataka se može naći više parova s istim ključem - svaka vrijednost je doprinos od neke druge stranice.

Linija 74 "spaja" sve parove s istim *ključem* tako što od više uređenih parova s istim *ključem* stvori jedan uređeni par s tim *ključem*, a *vrijednosti* zbroji. Time je zapravo izračunat ukupni primljeni doprinos svake od stranica i nastao je novi skup uređenih

parova oblika (*stranica*, *ukupni primljeni doprinos*).

Linija 75 dohvaća *vrijednosti* iz tih uređenih parova i na temelju njih računa nove *vrijednosti* i postavlja ih kao trenutne *vrijednosti*. Nove *vrijednosti* su zapravo novi rangovi svake od stranica. Sada se u varijabi `rangovi` nalaze uređeni parovi (*stranica*, *rang*) .

Postupak od linije 52 do linije 76 ponavljamo potrebni broj puta kako bi se rang svake od stanica postavio na što ispravniju vrijednost. Kao što je već rečeno, u praksi je to desetak puta pa je tako i u ovom primjeru.

Kada su izračunati rangovi svake od stanica, sve što još treba je ispisati podatke. Kako bi se uštedilo na vremenu (ponovno računanje rangova svaki puta kada treba raditi nešto s tim rangovima), dohvaćamo sve podatke iz skupa podataka `rezultati` i spremamo ih u varijablu `rezultatiFinal`. Taj dio je prikazan linijom 78.

Konačno, u `for` petlji iteriramo po elementima liste `rezultatiFinal` te ispišemo ime i rang stranice. Na kraju zatvorimo *SparkContext*.

## 3.2. Skupovi podataka kao uređeni parovi

U primjeru 3.1 već je uveden pojam skupa podataka koji je predstavljen uređenim parom (*ključ*, *vrijednost*). U ovoj tehnologiji takav skup podataka nije rijetkost i zbog toga postoji niz transformacija nad upravo tim oblikom skupa podataka, a neke od njih su već objašnjene u potpoglavlju 3.1.2. Osnovne informacije mogu se pronaći u tablicama 3.8 i 3.9, a za više informacija, konzultirati službenu dokumentaciju [1].

Često postoji potreba za transformacijom koja je kombinacija dva skupa podataka. Primjerice, spajanje tablica iz baze podataka. Te stvari su moguće i ovdje, a samo neke od transformacija prikazane su u tablici 3.9. Osim transformacija, postoje i akcije koje je moguće izvoditi nad otpornim raspodijeljenim skupom podataka koji je predstavljen kao uređeni par, ali radi formata ovog rada, nije napisana još jedna tablica gdje bi te akcije bile objašnjene. Za informacije o akcijama, konzultirati [1].

Ono što je još često potrebno, a nije već spomenuto jest sortiranje podataka. Tehnologija *Apache Spark* omogućuje i sortiranje uređenih parova po *ključu*. Transformacija se naziva `sortByKey` i primjer poziva je `rdd.sortByKey()`. Prima argument treba li sortirati uzlazno (podrazumijevana vrijednost) ili silazno. Dodatno, moguće je i predati komparator koji će usporediti ključeve.

**Tablica 3.8:** Neke od transformacija nad jednim skupom podataka koji je predstavljen uređenim parovima. Primjer: {(1, 2), (3, 4), (3, 6)}.

| Transformacija | Opis | Primjer korištenja | Rezultat |
|---|---|---|---|
| `reduceByKey(`*`func`*`)` | Kombinira vrijednosti s istim *ključem*. | rdd.reduceByKey((x,y) -> x + y) | {(1, 2), (3, 10)} |
| `groupByKey()` | Grupira vrijednosti s istim *ključem*. | rdd.groupByKey() | {(1, [2]), (3, [4, 6])} |
| `mapValues(`*`func`*`)` | Primjena funkcije nad svakom *vrijednosti* u uređenom paru, *ključ* ostaje nepromijenjen. | rdd.mapValues( x -> x + 10) | {(1, 12), (3, 14), (3, 16)} |
| `keys()` | Vraća novi skup podataka koji se sastoji od svih *ključeva*. | rdd.keys() | {1, 3, 3} |
| `values()` | Vraća novi skup podataka koji se sastoji od svih *vrijednosti*. | rdd.values() | {2, 4, 6} |

## 3.3. Čitanje i spremanje podataka

U ranijim primjerima prikazano je kako ostvariti čitanje podataka iz tekstualne datoteke kao i kako zapisati rezultat u neku tekstualnu datoteku. Neki od podržanih oblika tekstualnih datoteka su JSON, CSV i TSV oblik. Tehnologija *Apache Spark* omogućuje i puno više. Između ostalog, moguće je čitati i pisati direktno u bazu podataka (Cassandra, HBase, Elasticsearch, JDBC), strukturirane izvore uključujući JSON i Apache Hive, a isto tako moguće je čitati i podatke koji su spremljeni na raspodijeljenom sustavu datoteka poput NFS, HDFS ili Amazon S3.

**Tablica 3.9:** Neke od transformacija nad dva skupa podataka koji su predstavljeni uređenim parovima. Primjer: rdd = (1, 2), (3, 4), (3, 6), drugi = (3,9).

| Transformacija | Opis | Primjer korištenja | Rezultat |
|---|---|---|---|
| `subtractByKey` | Briše parove koji imaju iste *ključeve*. | rdd.subtractByKey(drugi) | {(1, 2)} |
| `join` | Obavlja operaciju *prirodnog spajanja* (engl. *inner join*) | rdd.join(drugi) | {(3, (4, 9)), (3, (6, 9))} |

Datoteke u JSON obliku se učitaju poput tekstualne datoteke, a zatim se parsira JSON format. Za parsiranje JSON formata se u pravilu koriste gotove biblioteke, a popularna biblioteka za Javu naziva se *Jackson*. Više informacija o tim bibliotekama može se pronaći na poveznici [1].

Više informacija o čitanju i spremanju podataka potražiti u poglavlju *Loading and saving your data* [2].

## 3.4. Dijeljene varijable

Funkcija koju predamo metodi `map()` vidi varijable koje su definirane u *pozivajućem programu*, ali svaki *zadatak* (engl. *task*) koji se vrti na *grozdu* dobije svoju kopiju vrijednosti tih varijabli, a nove vrijednosti tih varijabli nisu propagirane *pozivajućem programu*. Riješenje ovog problema su dijeljene varijable (engl. *shared variables*). Ime su dobile po tome što je njihova *ispravna* vrijednost vidljiva svuda u grozdu.

### 3.4.1. Odašiljatelji

Ukoliko je potrebno dohvaćati vrijednost neke varijable na cijelom grozdu, potrebna je *varijabla odašiljatelj* (engl. *broadcast variable*). Ta varijabla se pohranjuje u memoriji i zbog toga ne smije zauzimati puno prostora. Također, takva varijabla je nepromjenjiva (engl. *immutable*). Moguće je slati i promjenjivu varijablu, ali ta promjena će biti jedino vidljiva na tom čvoru koji je promjenu napravio, a takvo ponašanje u pravilu nije poželjno.

[1] `http://geokoder.com/java-json-libraries-comparison`

Kao što je navedeno u [1], *odašiljatelji* se mogu koristiti kao nepromjenjive pregledne tablice (engl. *lookup table*). Pretpostavimo da radimo na aplikaciji koja u bazi ima dvije tablice, tablicu koja pamti identifikator i ime naselja te tablicu koja pamti identifikator korisnika koji se prijavio i identifikator naselja gdje se prijavio. Primjer takvih tablica su tablice 3.10 i 3.11.

**Tablica 3.10:** Tablica `naselja`.

| id | ime |
|---|---|
| 1 | Centar |
| 2 | Maksimir |
| 3 | Jordanovac |
| 4 | Siget |
| 5 | Dubrava |

Zadatak je saznati ime naselje za svaku od prijava. Pretpostavka je da postoji jako puno prijava, a ne previše naselja tj. prijave su u nekoj velikoj tablici, a naselja su u maloj tablici. Pri obavljanju klasične transformacije `join`, podatci obje tablice bi putovali svuda po mreži i to bi uzelo puno vremena i mrežnih resursa. Ukoliko je potrebno napraviti `join` male i velike tablice, bolje rješenje je malu tablicu (koja stane u memoriju i ne mijenja se) postaviti kao varijablu *odašiljatelj*. Na taj način, jedino će mala tablica putovati preko mreže. U ovom primjeru, mala tablica je tablica `naselja` 3.10, a velika tablica je tablica `prijave` 3.11.

Varijable koje su postavljene kao *odašiljatelj* preko mreže se prenose protokolom *bittorrent*. To znači da što više čvorova treba tu varijablu, ta varijabla se brže širi po mreži[2].

```
1 package hr.fer.zemris.naprednoProgramiranje;
2
3 import java.util.Map;
4
5 import org.apache.spark.SparkConf;
6 import org.apache.spark.api.java.JavaPairRDD;
7 import org.apache.spark.api.java.JavaSparkContext;
8 import org.apache.spark.broadcast.Broadcast;
9
```

[2]Više informacija o bittorent protokolu na stranici `http://www.bittorrent.org/`

```
10 import scala.Tuple2;
11
12 public class Primjer5 {
13   public static void main(String[] args) {
14     // Inicijalizacija SparkContext-a.
15     SparkConf conf = new SparkConf()
16          .setMaster("local")
17          .setAppName("Razasiljane varijable");
18
19     try (JavaSparkContext sc = new JavaSparkContext(conf)) {
20
21       // inicijalizacija naselja
22       JavaPairRDD<String, String> naseljaRDD = sc
23         .textFile("tblNaselja.txt")
24         .mapToPair(
25           redakTablice -> {
26             String[] redak = redakTablice.split(",");
27             return new Tuple2<String, String>(
28               redak[0], redak[1]
29               );
30           }
31         );
32
33       // prijave
34       JavaPairRDD<String, String> prijaveRDD = sc
35            .textFile("tblPrijave.txt")
36            .mapToPair(redakTablice -> {
37              String[] redak = redakTablice.split(",");
38              return new Tuple2<String, String>(
39                redak[0], redak[1]
40                );
41              }
42            );
43
44       // odasiljanje read only varijable "naselja" kroz grozd
45       final Broadcast<Map<String, String>> naselja =
46         sc.broadcast(naseljaRDD.collectAsMap());
47
```

```
48
49          JavaPairRDD<String, Tuple2<String, String>>
                imenovanePrijave = prijaveRDD
50                .mapToPair(
51                  prijava -> new Tuple2<String, Tuple2<String,
                        String>>(
52                    prijava._1,
53                    new Tuple2<String, String>(
54                      prijava._2,
55                      naselja.getValue().get(prijava._2)
56                    )
57                  )
58                );
59          //ispis
60          for (Tuple2<String, Tuple2<String, String>> element :
61            imenovanePrijave.collect()) {
62            System.out.printf(
63              "Id prijave: %s \t Ime naselja: %s\n",
64                element._1,
65                element._2._2
66              );
67          }
68        }
69    }
70 }
```

**Primjer 3.2:** Korištenje odašiljatelja.

U primjeru 3.2 pokazano je korištenje odašiljatelja. Osim toga, u tom primjeru je *SparkContext* inicijaliziran u *try-catch* bloku naredbi. Budući da *SparkContext* implementira sučelje `Closeable`, to je garancija da ako ga otvorimo *try-catch* inicijalizacijskom bloku da će biti i pravovremeno zatvoren.

Program čita iz datoteke `tblNaselja.txt` podatke o naseljima koji su zapisani u obliku *id_naselja, ime_naselja*. Podatci o prijavama imaju oblik *id_prijave, id_naselja* i čitaju se iz datoteke `tblPrijave.txt`. Podatci u tim datotekama jednaki su podatcima iz tablica 3.10 i 3.11. Nakon što su podatci učitani, kreirani su skupovi podataka. Zatim se skup podataka koji predstavlja tablicu 3.10 pretvorio

u mapu gdje je *ključ* identifikator naselja, a *vrijednost* ime naselja. Takva mapa je *odaslana* svuda po mreži.

**Tablica 3.11:** Tablica `prijave`.

| id_korisnik | id_naselje |
|---|---|
| 112312 | 1 |
| 243242 | 2 |
| 254331 | 2 |
| 432115 | 1 |
| 567893 | 4 |
| 537893 | 4 |

Nakon toga, za svaku prijavu, kreiran je novi uređeni par koji za *ključ* ima identifikator prijave, a za vrijednost objekt `Tuple2` koji za prvu vrijednost ima identifikator nasilja, a za drugu vrijednost ime tog naselja. Razred `Tuple2` preuzet je iz programskog jezika Scala i predstavlja uređeni par (*vrijednost_1*, *vrijednost_2*). Shodno tome, prvoj vrijednosti se pristupa preko `element._1`, a drugoj vrijednost preko `element._2`.

Na kraju su ispisani svi dobiveni podatci, a rezultat je prikazan u primjeru 3.3.

```
Id prijave: 112312  Ime naselja: Centar
Id prijave: 243242  Ime naselja: Maksimir
Id prijave: 254331  Ime naselja: Maksimir
Id prijave: 432115  Ime naselja: Centar
Id prijave: 567893  Ime naselja: Siget
Id prijave: 537893  Ime naselja: Siget
```

**Primjer 3.3:** Identifikatori prijave i imena naselja odakle su prijave poslane.

### 3.4.2. Akumulatori

Ako se odašiljatelje gleda kao globalne *gettere*, *akumulatore* (engl. *accumulators*) se može gledati kao na globalne *settere*. Za razliku od odašiljatelja koji svim čvorovima dojavljuju vrijednost, akumulatori akumuliraju, odnosno dohvaćaju vrijednost svakog čvora. Ideja je jednostavna, ako je potrebno s više radnih čvorova mijenjati nekakvu globalnu varijablu, ta varijabla se postavi kao akumulator.

U datoteci `log.txt` pohranjeni su zapisi pristupanju na poslužitelj. Svaki zapis je u novom retku. Ukoliko u nekom retku postoji riječ `error`, taj zapis nije valjan.

Svaki drugi zapis je valjan. Zadatak je izbrojati koliko postoji valjanih zapisa i koliko postoji zapisa koji nisu valjani. Jedno od riješenja je koristiti dva akumulatora: jedan za valjane zapise i drugi za zapise koji nisu valjani. Programska implementacija tog rješenja dana je u primjeru 3.4.

```
1  package hr.fer.zemris.naprednoProgramiranje;
2
3  import org.apache.spark.Accumulator;
4  import org.apache.spark.SparkConf;
5  import org.apache.spark.api.java.JavaRDD;
6  import org.apache.spark.api.java.JavaSparkContext;
7
8  public class Primjer6 {
9    public static void main(String[] args) {
10     SparkConf conf = new SparkConf()
11         .setMaster("local")
12         .setAppName("Koristenje akumulatora");
13     try (JavaSparkContext sc = new JavaSparkContext(conf)) {
14       // inicijalizacija naselja
15       JavaRDD<String> zapisi = sc.textFile("log.txt");
16
17       // inicijalizacija akumulatora
18       Accumulator<Integer> valjaniZapisi = sc.accumulator(0);
19       Accumulator<Integer> neValjaniZapisi =
             sc.accumulator(0);
20
21       // obrada podataka
22       for (String redak : zapisi.collect()) {
23         if (redak.contains("error")) {
24           neValjaniZapisi.add(1);
25         } else {
26           valjaniZapisi.add(1);
27         }
28       }
29       // ispis rezultata
30       System.out.printf("Valjanih zapisa ima %d\n",
31           valjaniZapisi.value());
32       System.out.printf("Nevaljanih zapisa ima %d\n",
```

```
33                neValjaniZapisi.value());
34        }
35    }
36 }
```

**Primjer 3.4:** Korištenje akumulatora.

Primjer 3.4 učita novi skup podataka, inicijalizira akumulatore te za svaki redak, ovisno sadrži li redak riječ *error* ili ne, poveća vrijednost odgovarajućeg akumulatora.

U primjeru 3.4 korišteni su cjelobrojni akumulatori. Osim *Integer* tipa podataka, postoje akumulatori i za *Long*, *Double* te *Float* tip podataka. Ukoliko je potrebno, moguće je kreirati i akumulator za svoj vlastiti tip podataka. Za više informacija, konzultirati [4].

# 4. Zaključak

Danas već postoji više podataka u digitalnom obliku nego ikada prije. U budućnosti se pretpostavlja da će ih biti još i više. Tehnologija *Apache Spark* nudi mogućnost obrade velikog skupa podataka. U radu je pokazano da ima bogato programsko sučelje za programski jezik Java i da se u svega nekoliko naredbi može dosta dobro obraditi te podatke.

Kao nastavak ovog rada predlaže se istražiti kako instalirati i koristiti ovu tehnologiju na grozdu. Također bi bilo dobro i detaljnije istražiti te opisati svaki od gradivnih elemenata od kojih se ova tehnologija sastoji. Ovo je tehnologija koja je svakim danom sve popularnija i u budućnosti bi mogla imati široku primjenu.

# LITERATURA


[1] Službena dokumentacija projekta *Apache Spark* `http://spark.apache.org/docs/latest/` , posjećeno 1.5.2016

[2] Holden Karau, Andy Konwinski, Patrick Wendell i Matei Zaharia *Learning Spark, lighting-fast*, O'Reilly 2015.

[3] Marko Čupić, *Programiranje u Javi*, verzija 0.3.26

[4] Zaharia, Matei, et al. "Spark: Cluster Computing with Working Sets." *HotCloud 10* (2010): 10-10.

[5] Xing, Wenpu, i Ali Ghorbani. "Weighted pagerank algorithm." *Communication Networks and Services Research, 2004. Proceedings. Second Annual Conference on.* IEEE, 2004.

# Dodaci

# A. Instalacija

Ovdje je prikazana instalacija na operacijskom sustavu *Ubuntu 15.10*. Ovo nije veliko ograničenje jer se iz ovih uputa može zaključiti i kako instalacija ide za neki drugi operacijski sustav.

Za početak je potrebno imati instaliranu Javu, a je li Java instalirana na računalu se može provjeriti tako što se u naredbenom retku unese sljedeća naredba:
`java -version`. Kao rezultat se dobije trenutno instalirana verzija Jave. Ukoliko Java nije instalirana, potrebno ju je najprije instalirati. Više informacija o instalaciji se može pronaći u [3].

Jednom kada je Java instalirana, sve što treba napraviti je otići na službene stranice: `https://spark.apache.org/downloads.html`, odabrati najnoviju verziju (Za vrijeme pisanja ovog rada to je verzija *1.6.1 (Mar 09 2016*), izabrati odgovarajući paket te pokrenuti dohvaćanje odgovarajuće *.tgz* arhive. Najjednostavnije je odabrati neki *pre-built* paket, primjerice *Pre-built for Hadoop 2.6 and later*. Daljnji koraci instalacije su napisani pod pretpostavkom da je dohvaćena ta verzija paketa. Moguće je instalirati i *Source code* varijantu paketa, ali taj postupak instalacije ovdje nije opisan.

Nakon što je dohvaćena odgovarajuća arhivu, potrebno ju je raspakirati.
Raspakiranje arhive moguće je napraviti preko naredbe:

```
$ tar -xvf spark-1.6.1-bin-hadoop2.6.tgz
```

Nakon toga, dobra je praksa premjestiti instalaciju u neki prikladniji direktorij. Tako nešto može se napraviti na sljedeći način:

```
$ mv Downloads/spark-1.6.1-bin-hadoop2.6 faks/spark/
```

Ovim korakom je instalacija završena.

Kako bi bili sigurni da je instalacija uspješna, potrebno je pozicionirati se u *bin* direktorij te u terminalu upisati `./spark-shell`. Ispis bi trebao biti sličan ovome:

```
mmatak@martins-beast:~/faks/spark/bin$ ./spark-shell
```

```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\ version 1.6.1
      /_/

Using Scala version 2.10.5 (OpenJDK 64-Bit Server VM, Java
   1.8.0_91)
Type in expressions to have them evaluated.
Type :help for more information.
scala>
```

Ukoliko se prikaže greška vezana uz *sqlContext*, to nije razlog za brigu. U ovom trenutku to nije važno. Radi ljepšeg formata ovog rada, u gornjem ispisu izbrisana su upozorenja (engl. *warnings*).

# B. Postavljanje datoteke pom.xml

Datoteka koja je potrebna kako bi Maven ispravno dohvatio artifakt `spark-core` bi trebala izgledati slično kao što je dano u nastavku. Za ispravno funkcioniranje, trebala bi se zvati `pom.xml`.

```
<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>hr.fer.zemris</groupId>
  <artifactId>Prvi-programi</artifactId>
  <version>0.0.1-SNAPSHOT</version>
  <name>Prvi programi</name>
  <description>Prvi programi u Spark-u</description>
  <dependencies>
   <dependency>
     <groupId>org.apache.spark</groupId>
     <artifactId>spark-core_2.10</artifactId>
     <version>1.6.1</version>
   </dependency>
  </dependencies>
</project>
```

## Obrada podataka tehnologijom Apache Spark


### Sažetak

U ovom radu opisano je kako koristiti tehnologiju *Apache Spark*. Objašnjeno je koji su glavni elementi te tehnologije i čemu pojedini element služi. Napisano je i nekoliko primjera gdje se vidi kako se tehnologija koristi kroz programski jezik Java. Detaljno je opisan algoritam *PageRank* radi boljeg pojašnjenja programskog sučelja koje ova tehnologija nudi. Također je objašnjeno i kako pravilno postavljati vrijednosti varijabli na grozdu.



**Ključne riječi:** Apache Spark, otporni raspodijeljeni skup podataka, transformacije, akcije, akumulatori, odašiljatelji


## Data processing with technology Apache Spark


### Abstract

In this paper it is described how to use technology *Apache Spark*. It is explained of which main elements Spark consists of and what is a purpose of each element. It is also written a few examples where is shown how to use this technology through Java programming language. *PageRank* algorithm is briefly described and this way is application programming interface for Spark in Java also explained. It is also shown how to properly use variables on cluster.



**Keywords:** Apache Spark, Resilient distributed dataset, transformations, actions, accumulators, broadcast variables