УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ

Давид Гавриловић

ДИСТРИБУИРАНА ОБРАДА ГЕОПРОСТОРНИХ ПОДАТАКА

мастер рад

Београд, 2022.

Ментор:

др Мика Микић, редован професор
Универзитет у Београду, Математички факултет

Чланови комисије:

др Ана Аничић, ванредни професор
University of Disneyland, Недођија

др Лаза Лазић, доцент
Универзитет у Београду, Математички факултет

Датум одбране: 15. јануар 2016.

некоме

Наслов мастер рада: Дистрибуирана обрада геопросторних података

Резиме: **TODO** Фијуче ветар у шибљу, леди пасаже и куће иза њих и гунђа у оџацима. Ниџо, чежњиво гледаш фотељу, а Ђура и Мика хоће позицију себи. Људи, јазавац Џеф трчи по шуми глођући неко сухо жбуње. Љубави, Олга, хајде пођи у Фуџи и чут ћеш њежну музику срца. Боја ваше хаљине, госпођице Џафић, тражи да за њу кулучим. Хаџи Ђера је заћутао и бацио чежњив поглед на шољу с кафом. Џабе се зец по Хомољу шуња, чувар Јожеф лако ће и ту да га нађе. Оџачар Филип шаље осмехе туђој жени, а његова кућа без деце. Бутић Ђуро из Фоче има пун џак идеја о слагању ваших жељица. Џајић одскочи у аут и избеже ђон халфа Пецеља и његов шамар. Пламте оџаци фабрика а чађаве гује се из њих дижу и шаљу ноћ. Ајшо, лепото и чежњо, за љубав срца мога, дођи у Хаџиће на кафу. Хучи шума, а иза жутог џбуна и пања ђак у цвећу деље сеји фрулу. Гоци и Јаћиму из Бање Ковиљаче, флаша џина и жеђ падаху у исту уру. Џаба што Фећа чупа за косу Миљу, она јури Живу, али њега хоће и Даца. Док је Фехим у џипу журно љубио Загу Чаћевић, Циле се ушуњао у ауто. Фијуче кошава над оџацима а Иља у гуњу журећи уђе у суху и топлу избу. Боже, џентлмени осећају физичко гађење од прљавих шољица! Дочепаће њега јака шефица, вођена љутом срџбом злих жена. Пази, гецо, брже однеси шефу тај ђавољи чек: њим плаћа цех. Фине џукце озлеђује бич: одгој их пажњом, стрпљивошћу. Замишљао би кафеџију влажних прстића, црњег од чађи. Ђаче, уштеду плаћај жаљењем због џиновских цифара. Џикљаће жалфија између тог бусења и пешчаних дворaца. Зашто гђа Хаџић лечи живце: њена љубав пред фијаском? Јеж хоће пецкањем да вређа љубичастог џина из флаше. Џеј, љубичаст зец, лаже: гаћаће одмах поквашен фењер. Плашљив зец хоће јефтину динју: грожђе искамчи џабе. Џак је пун жица: чућеш тад свађу због ломљења харфе. Чуј, џукац Флоп без даха с гађењем жваће стршљена. Ох, задњи шраф на џипу слаб: муж гђе Цвијић љут кочи. Шеф џабе звиждуће: млађи хрт јаче кљуца њеног пса. Одбациће кавгаџија плаштом чађ у жељезни фењер. Дебљи кројач: згужвах смеђ филц у тањушни џепић. Џо, згужваћеш тихо смеђ филц најдебље крпењаче. Штеф, бацих сломљен дечји зврк у џеп гђе Жуњић. Дебљој згужвах смеђ филц — њен шкрт џепчић.

Кључне речи: анализа, геометрија, алгебра, логика, рачунарство, астрономија

Садржај

Глава 1

Увод

увод о свему у раду
TODO

Глава 2

Програмски језик Скала

Скала (енг. Scala) је виши програмски језик заснован на функционалној и објектно оријентисаној парадигми. Име је добила од енглеске речи *scalable* јер је дизајнирана тако да се развија са потребама корисника. Има широк спектар примена и може се користити за писање једноставних скрипти али и у изградњи великих и комплексних система. [8]

Настала је 2001. на швајцарском федералном институту за технологију у Лозани (фра. *École Polytechnique Fédérale de Lausanne*) и њен творац је Мартин Одерски (енг. *Martin Odersky*). Прва званична верзија је изашла 20. јануара 2004. године.

Данас је широко распрострањена и користе је велике корпорације, као што су *Twitter*, *Google* и *Apple*. Поред тога, веома је заступљена у заједници отвореног кода у пројектима као што су *Apache Spark*, *Apache Kafka*, *Apache Flink* и *Akka*.

2.1 Особине језика *Scala*

Scala је спој две парадигме, објектно оријентисане и функционалне, па стога поседује велики број особина. Поред тога, компајлира се на исти начин као и језик Јава, са којим постоје одређене сличности.

Објектно оријентисан и функционалан језик

Скала је у потпуности објектно оријентисан језик. То значи да је свака вредност која се дефинише објекат, као и да је свака акција која се позива

метод. На пример, уколико се врши одузимање два цела броја, позива се метод назван ″−″ (минус). Тај метод је дефинисан у класи која представља целе бројеве, *Int*. [8]

Поред тога што је објектно оријентисан језик, *Scala* је и функционалан језик. Функционално програмирање је засновано на два принципа. Први је да су функције вредности првог реда. То значи да се функције посматрају на исти начин као и други типови, на пример целобројни или ниска. Такође, функције је могуће прослеђивати другим функцијама, функције могу бити повратна вредност неке друге функције и функције се могу складиштити у променљивама.

Други принцип је да функције које се позивају немају бочне ефекте. Једна функција има улогу само да пресликава улаз у одговарајући излаз. То значи да ће сваки позив једне функције са истом вредношћу улазних аргумената, увек резултовати истом излазном вредношћу, независно од тога када се функција позива током извршавања програма. Другачији назив за ову особину је транспарентост референци. [8]

Из овога произилази да функционални језици користе имутабилне структуре података. То су такве структуре за које важи да се подаци унутар њих не мењају. Уколико до промене мора доћи, сама структура се не мења, већ се од ње конструише тотално нова, са измењеним вредностима. [8]

Међутим, Скала није чисто функционалан језик, што значи да се ипак може дефинисати функција која поседује бочне ефекте или се могу користити структуре података које се могу мењати. Поред тога, Скала омогућава писање функционалног кода без бочних ефеката и са имутабилним структурама података, па се може користити и на тај начин. [8]

Повезаност са језиком Јава

Scala је конструисана тако да се компајлира у Јавин JBM бајткôд (енг. *Java JVM bytecode*). То значи да *Scala* може користити Јава класе, методе и типове. На пример, Скалин објектни целобројни тип у својој имплементацији користи примитивни еквивалент из Јаве. Поред тога, Скала може користити Јава кôд и обогатити га на неки начин, као на пример додавањем неке методе у већ постојећу класу. Напоменимо и то да је време извршавања Скала програма приближно извршавању Јава програмима. [8]

Међутим, иако се компајлирају на исти начин, програми написани у језику Скала често садрже мањи број линија од оних написаних у језику Јава. У неким случајевима се очекује да је кôд чак дупло краћи. Краћи програми доводе до тога да је кôд лакше писати и разумети, али и до мање вероватноће прављења грешака. [8]

Један од многих примера како Скала смањује број линија у односу на Јаву је приказан у кодовима 2.1 и 2.2 који представљају начине декларисања класе у та два програмска језика.

```
// Java
class MyClass {
  private int index;
  private String name;

  public MyClass(int index, String name) {
    this.index = index;
    this.name = name;
  }
}
```

Кôд 2.1: Декларација класе у језику Јава

```
// Scala
class MyClass(index: Int, name: String)
```

Кôд 2.2: Декларација класе у језику Scala

У Скали се не мора декларисати посебна конструктор фукција, што доводи до смањења броја линија кода. [8]

Статичка типизираност

Статичка типизираност значи да се типови променљивих закључују за време компилације програма. То је супротно од динамичке типизираности која то чини за време извршавања. Оба приступа имају своје предности и мане. Скала је статички типизиран језик и поседује веома напредан систем типова.

То доноси предности које доводе до лакшег откривања грешака приликом писања кода. На пример, у статички типизираним програмима се током

компилације сазнаје да ли је примењена нека операција на тип над којим та операција није дозвољена. Поред тога, статичка типизираност чини рефакторисање кода поузданијим. На пример, након измене метода се са сигурношћу може рећи да се повратни тип није променио. [8]

Скала није само статички типизиран језик већ је и језик који аутоматски закључује типове у току компилације. На пример, када се декларише нека променљива целобројног типа, нема потребе назначити и њен тип, пошто га компајлер може аутоматски одредити. То значи да се следеће две линије кода (пример 2.3) понашају еквивалентно. Исто важи и за било који други Скала тип, не само за целобројни.

```
// primer 1
val x: Int = 10

// primer 2
val x = 10
```

Кôд 2.3: Декларација типова

Скала програмер не мора експлицитно да наводи типове, али је то често пожељно због тога што се на тај начин осигурава да ће кôд заправо користити тип који му је намењен, али ће и побољшати читљивост и допринети документацији. [8]

2.2 Скала интерпретер

Скала је језик који се може интерпретирати. Да би се покренуо Скала интерпретер потребно је покренути команду *scala*.

```
$ scala
Welcome to Scala 2.13.6
Type in expressions for evaluation. Or try :help.

scala>
```

Кôд 2.4: Скала интерпретер

Након што се унесе кôд у интерпретер и притисне ентер, покреће се извршавање написаног кода и излаз се приказује у конзоли.

```
scala> 20 + 100
val res0: Int = 120
```

Кôд 2.5: Пример кода у интерпретеру

Излаз покренуте команде је аутоматски генерисана промењљива названа **res0** типа *Int* у којој ће се налазити резултат унетог израчунавања. Новонастала промењљива се може користити у наставку извршавања. [8]

```
scala> res0 + 100
val res1: Int = 220
```

Кôд 2.6: Коришћење резултатских промењљивих

Уколико је потребно само исписати вредност у конзоли без креирања нове промењљиве може се користити функција *print()*.

```
scala> print(20 + 100)
120

scala> print("Hello!")
Hello!
```

Кôд 2.7: Функција print

2.3 Типови

Сви примитивни типови Јаве имају свој одговарајући еквивалент у Скали и када се типови у Скали компајлирају у Јавин бајткôд, превешће се баш у те типове. На пример, логички тип у Скали, *scala.Boolean* је еквивалент Јавином примитивном типу *boolean*. Исто важи и за друге примитивне типове Јаве попут *Int*, *Float* и *Double*. [8]

Поред њих постоје и уграђени сложени типови попут ниске (*String*), н-торке (*Tuple*), низа (*Array*) и других. Како је Скала објектно оријентисан језик, могу се дефинисати и додатни типови уколико за тим има потребе, али о томе више речи у секцији 2.7.

Сваки тип, долази са скупом оператора који се на тај тип могу применити. Скала је написана тако да је сваки оператор заправо један метод дефинисан у класи која представља тип. Постоје различите врсте оператора попут аритметичких, логичких и битовских.

2.4 Промењиве

Постоје две врсте промењивих које се дефинишу кључним речима *var* и *val*. Разликују се по томе што се вредност промењивих дефинисаних са *val* не може мењати, док је код оних дефинисаних са *var* то могуће, све док је нова додељена вредност компатибилног типа. [8]

У наставку (примери 2.8, 2.9 и 2.10) су приказани примери дефиниција промењивих у Скала интерпретеру.

```
scala> val x = 10
val x: Int = 10

scala> x = 20
         ^
       error: reassignment to val
```

Кôд 2.8: Додељивање нове вредности val промењивима

```
scala> var x = 10
var x: Int = 10

scala> x = 20
// mutated x

scala> x
val res0: Int = 20
```

Кôд 2.9: Додељивање нове вредности var промењивима

```
scala> var x = 10
var x: Int = 10

scala> x = "some string"
           ^
       error: type mismatch;
        found   : String("some string")
        required: Int
```

Кôд 2.10: Додељивање некомпатибилног типа

2.5 Контрола тока

Скала поседује уграђене стандардне наредбе за контролу тока, *if* за гранање, *while* за петље и *for* и *foreach* за итерирање кроз колекције. У наставку су те наредбе приказане у скалиној синтакси. [8]

```
if (bool izraz) {
  // izraz je evalueiran true
} else {
  // izraz je evaluiran false
}

while (bool izraz) {
  // dok se izraz ne evaluira false
}

for (element <- kolekcija) {
  // operacije nad elementom
}

kolekcija.forach(funkcija koje se poziva za svaki element
   kolekcije)
```

Кôд 2.11: Контрола тока

2.6 Функције

Скала делом припада функционалној парадигми па су стога функције веома битан део језика. Функција се дефинише кључном речи *def* након које редом следе име функције, опциона листа њених аргумената са њиховим типовима раздвојених зарезом, тип повратне вредности функције, знак = и на крају тело функције. Пример дефиниције је приказан у коду 2.12.

```
def imeFunkcije(argument1: tip1, ...): povratni_tip = {
  // telo funkcije
```

```
}
```

Кôд 2.12: Дефиниција фунције у скали

У наставку је приказана функција која сабира два целобројна броја и враћа добијени резултат.

```
def saberi(x: Int, y: Int): Int = {
  x + y
}
```

Кôд 2.13: Пример функције

Последња линија тела функције ће увек бити њена повратна вредност али се поред тога она може назначити и наредбом *return*. Уколико се функција састоји од само једне линије кода, могу се изоставити витичасте заграде које означавају почетак и крај тела фунције. Поред тога, због закључивања типова се може изоставити и тип повратне вредности. Дакле, функција *saberi()* из претходног примера се краће може записати на следећи начин:

```
scala> def saberi(x: Int, y: Int) = x + y
def saberi(x: Int, y: Int): Int

scala> saberi(40, 2)
val res0: Int = 42
```

Кôд 2.14: Краћи запис функције saberi()

Тип повратне вредности се у неким случајевима ипак не сме изоставити. На пример, када се користи рекурзија. Такође, функција не мора да враћа никакву вредност. У том случају је повратни тип означен са *Unit*. [8]

Све функције су вредности првог реда у Скали па имају и свој тип. Тип функције је представљен заградама у којима се налазе типови њених аргумената након којих следи знак => праћен типом повратне вредности. Тип функције *saberi()* која поседује два аргумента типа *Int*, као и исти повратни тип ће бити:

```
(Int, Int) => Int
```

Кôд 2.15: Тип функције saberi()

Експлицитно навођење типова дозвољава декларацију функција вишег реда, функција које као аргументе имају друге функције. Пример 2.16 при-

казује функцију која као аргумент има функцију која има два аргумента и повратну вредност типа *Int*

```
scala> def visiRed(f: (Int, Int) => Int, x: Int, y: Int) =
   {
  f(x, y)
}
def visiRed(f: (Int, Int) => Int, x: Int, y: Int): Int
```

Кôд 2.16: Функција вишег реда

Тип овог аргумента одговара типу функције *saberi()*, па се она може проследити ново написаној функцији.

```
scala> visiRed(saberi, 100, 200)
val res0: Int = 300
```

Кôд 2.17: Прослеђивање функције функцији

Све функције које су до сада приказане су поседовале идентификатор, односно име. Међутим, то није неопходно и могуће је дефинисати функцију без имена. Такве функције се називају ламбда функције (енг. *Lambda functions*).

Оне се обично користе када је нека функција потребна само једном, на пример у неком изразу, и не позива се никад више у коду. Декларишу се тако што се у заградама наводи низ аргумената са типовима, знак => и након тога повратна вредност. Пример ламбда функције која сабира два броја је приказан у наставку.

```
scala> (x: Int, y: Int) => x + y
val res0: (Int, Int) => Int = $Lambda$2582/1961424035@2207
   eb9f
```

Кôд 2.18: Пример ламбда функције

Ламбда функције се могу проследити функцијама вишег реда, па претходно дефинисана функција *visiRed()* може бити позвана на следећи начин:

```
scala> visiRed((x: Int, y: Int) => x + y, 100, 200)
val res0: Int = 300
```

Кôд 2.19: Прослеђивање ламбда функције другој функцији

У овом примеру, функција *saberi()* је замењена ламбда функцијом истог понашања, што није довело до промене коначног резултата.

2.7 Објектно оријентисана парадигма

У овој секцији ће детаљније бити описана објектно оријентисана парадигма језика Скала.

Класе

Као и у Јави, у Скали класа представља нацрт преко ког се производе објекти. Да би се од класе креирао објекат, користи се кључна реч *new*.

```
scala> class MyClass {
     |
     | }

scala> val mc = new MyClass
val mc: MyClass = MyClass@e700eba
```

Кôд 2.20: Дефиниција и инстанцирање класе у Скали

Унутар класе се дефинишу поља (енг. *fields*) и методе (енг. *methods*), који се заједно једним именом називају чланови (енг. *members*). Поља су промењиве које се дефинишу са *val* или *var* док су методи функције које описују неко понашање и дефинишу се на исти начин као и обичне функције. [8]

```
scala> class MyClass {
     |   val field = 0
     |
     |   def method() = print(field)
     | }

scala> val mc = new MyClass
val mc: MyClass = MyClass@e700eba

scala> mc.field
mval res0: Int = 0

scala> mc.method()
```

```
0
```

Кôд 2.21: Чланови класе

Сваком члану се додељује једно правило приступа којим се одређује опсег из ког се том члану може приступити. У Скали постоје три правила приступа и то су:

- *private*, видљивост унутар класе
- *protected*, видљивост унутар класе али и унутар класа које наслеђују ту класу
- *public*, подразумевана вредност која се не наводи. Овим члановима се може приступити и изван саме класе

Пример *private* приступа је приказан у коду 2.22. Сваки покушај приступа приватној променљивој ван класе ће резултовати грешком.

```
scala> class MyClass {
     |   private val field = 0
     |
     |   def method() = print(field)
     |   }
class MyClass

scala> val mc = new MyClass
val mc: MyClass = MyClass@6a6e9289

scala> mc.field
           ^
       error: value field in class MyClass cannot be
   accessed as a member of MyClass from class
```

Кôд 2.22: Пример private правила приступа

Поља се могу дефинисати ван тела класе, што је и Скалин стандард (кôд 2.23). Због тога се класа може написати уз помоћ мањег броја линија.

```
scala> class MyClass(private val field: Int = 0) {
     |     def method() = print(field)
```

```
     |   }

scala> val m = new MyClass
val m: MyClass = MyClass@5d8e4fa8
```

Кôд 2.23: Дефиниција поља ван тела класе

У претходном примеру, поље *field* поседује подразумевану вредност која ће се том пољу увек доделити приликом инстанцирања класе. Међутим, она се не мора навести и, уколико је то случај, пољима се мора експлицитно доделити вредност приликом инстанцирања. Пример је приказан у коду 2.24. [8]

```
scala> class MyClass(private val field: Int) {
     |     def method() = print(field)
     |   }

scala> val mc = new MyClass
                    ^
       error: not enough arguments for constructor MyClass:
    (field: Int): MyClass.
       Unspecified value parameter field.

scala> val mc = new MyClass(10)
val mc: MyClass = MyClass@7d332e20
```

Кôд 2.24: Инстанцирање класе без подразумеваних вредности поља

Наслеђивање

Наслеђивање се извршава на исти начин као у Јави, преко кључне речи *extends*. Инстанцирање поља надкласе из подкласе се дефинише у самој дефиницији наслеђивања, након *extends* (Пример 2.25). [8]

```
// nadklasa
scala> class MyClass(private val field: Int) {
     |     def method() = print(field)
     |   }
```

```
// podklasa
scala> class MyExtendedClass(
     |     private val field: Int,
     |     private val newField: Int
     | ) extends MyClass(field) // prosledjivanje vrednosti
    nadklasi
class MyExtendedClass

scala> val mec = new MyExtendedClass(10, 20)
val mec: MyExtendedClass = MyExtendedClass@42ba9b22
```

Кôд 2.25: Наслеђивање у Скали

Предефинисање чланова надкласе се врши на исти начин као у Јави, преко кључне речи *override*.

Апстрактне класе

Апстрактне класе се дефинишу коришћењем кључне речи *abstract* која се наводи пре речи *class* која означава класу, на исти начин као у Јави. [8]

```
scala> abstract class MyAbstractClass {
     |
     | }
class MyAbstractClass
```

Кôд 2.26: Апстрактна класа у Скали

Апстрактне класе се не могу инстанцирати, али се могу наследити од стране других класа.

```
scala> abstract class MyAbstractClass {
     |
     | }
class MyAbstractClass

scala> val mac = new MyAbstractClass
                 ^
```

```
       error: class MyAbstractClass is abstract; cannot be
   instantiated
```

Кôд 2.27: Инстанцирање апстрактне класе

Синглтон објекти

За разлику од Јаве, у Скали не постоје статичка поља. Уместо тога постоје синглтон објекти (енг. *singleton object*). Дефинишу се на исти начин као и класе, с тим што се користи кључна реч *object* уместо *class*. Добили су име по томе што представљају класу која има тачно једну инстанцу. Сви чланови објекта се могу посматрати као статички чланови у Јава класи. [8]

```
scala> object MyObject {
     |   def hello() = print("Hello from object")
     | }

scala> MyObject.hello()
Hello from object
```

Кôд 2.28: Коришћење синглтон објекта

Уколико објекат дели своје име са неком класом, а при томе се налазе у истом фајлу, тај објекат се назива објекат пратилац (енг. *companion object*). Паралелно, та класа се назива класа пратилац тог објекта. Класа или објекат који су пратиоци могу да приступе приватним члановима свог пратиоца (кôд 2.29). [8]

```
object Kvadrat {
    def izracunajPovrsinu(a: Int) = a * a
}

class Kvadrat(a: Int) {
    def povrsina = Kvadrat.izracunajPovrsinu(a)
}

scala> val k = new Kvadrat(10)
val k: Kvadrat = Kvadrat@6cb2d5ea
```

```
scala> k.povrsina
val res0: Int = 100
```

Кôд 2.29: Пример пратиоца

Метод *main()*

Да би се Скала апликација покренула потребно је дефинисати објекат који у себи садржи *main()* метод. Тај метод представља улазну тачку у сваку Скала апликацију. [8]

```
scala> object Main {
     |   def main(args: Array[String]): Unit = {
     |       print("Hello")
     |   }
     | }
```

Кôд 2.30: Пример main метода

Скала интерфејси

Основна јединица наслеђивања у Скали се назива *Trait*. Унутар њих се наводе поља и методи који се могу користити у класама које их имплементирају, односно наслеђују. Разлика између наслеђивања *trait*-а и класе је та што је дозвољено наследити једну класу, док је могуће наследити више од једног *trait*-а. Дефинишу се на исти начин као и класе с тим што се уместо кључне речи *class* користи реч *trait* (кôд 2.31). [8]

Унутар *trait*-а се декларишу и дефинишу поља и методи које класе које га имплементирају могу користити.

```
scala> trait MyTrait {
     |   def myMethod(): Unit
     | }
trait MyTrait
```

Кôд 2.31: Скала *trait*

Trait се додаје класи на исти начин као када се означава наследство, помоћу речи *extends*.

```
scala> trait MyTrait {
     |   val x = 10
     | }
trait MyTrait

scala> class MyClass extends MyTrait
class MyClass

scala> val mc = new MyClass
val mc: MyClass = MyClass@774189d0

scala> mc.x
val res0: Int = 10
```

Кôд 2.32: Додавање *trait*-а класи

Уколико класа којој се додељује *trait* већ наслеђује неку класу или неки други *trait*, додељивање се мора извршити преко кључне речи *with*. Сваки нови *trait* који се додаје у овом случају се мора додати након нове речи *with*. Примери су приказани у коду 2.33. [8]

```
scala> class MyExtendedClass extends MyClass with MyTrait1
   with MyTrait2
class MyExtendedClass

scala> class MyExtendedTraits extends MyTrait1 with
   MyTrait2
class MyExtendedTraits
```

Кôд 2.33: Наслеђивање више *trait*-ова

Скала *trait* се може користити и као тип, а вредност промењиве тог типа мора бити класа која наслеђује тај *trait*.

```
scala> trait MyTrait
trait MyTrait

scala> class MyClass extends MyTrait
class MyClass
```

```
scala> val mc: MyTrait = new MyClass
val mc: MyTrait = MyClass@339cedbb
```

Кôд 2.34: *Trait* као тип

Из наведених карактеристика се може закључити да је Скала *trait* веома сличан Јавином интерфејсу, са разликом да *trait* може садржати и дефиниције метода и поља, а не само декларације. Међутим, *trait* је више од тога и унутар њега се може урадити све што се може урадити унутар Скала класе.

Case класе

У језику Скала, поред стандардних, постоји још једна врста класе названа *case* класа. Постоје три разлике између ове врсте класа и обичних:

- Променљиве ове класе се инстанцирају без кључне речи *new*
- Свако поље ове класе мора имати префикс *val*
- *Case* класа садржи аутоматски генерисане методе ==, *toString()* и *hashCode()*

```
scala> case class MyCaseClass(val field: Int)
class MyCaseClass

scala> val mcc = MyCaseClass(100)
val mcc: MyCaseClass = MyCaseClass(100)

scala> mcc.toString
val res0: String = MyCaseClass(100)
```

Кôд 2.35: Пример коришћења *case* класа

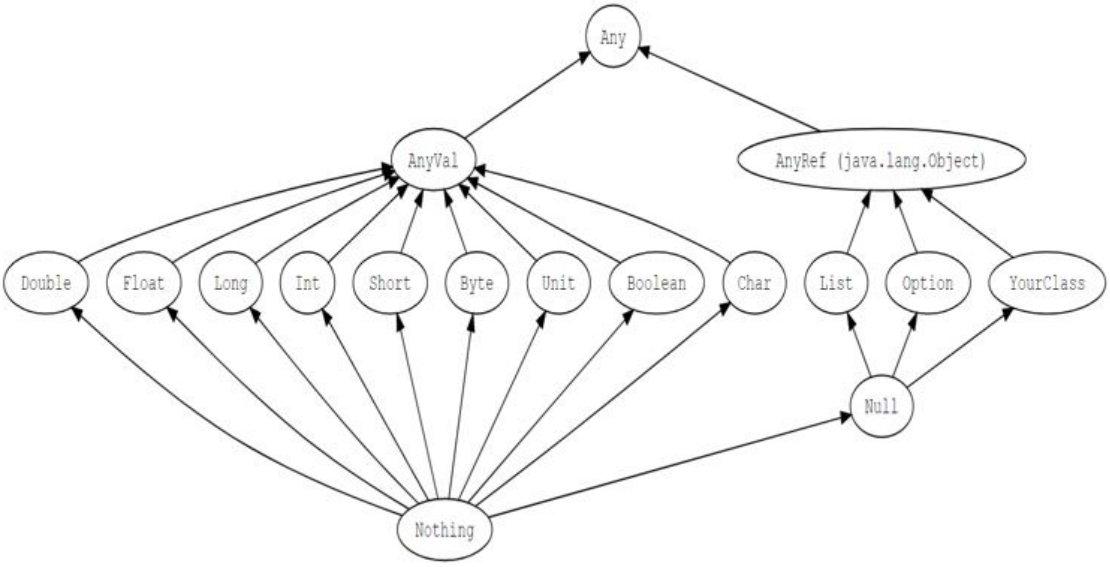
Једна од предности ових класа је та да се могу користити у конструкту специфичном за функционалне језике, поклапању образаца (енг. *pattern matching*), који ће бити детаљније описан у секцији 2.9. [8]

Хијерархија класа

У Скали, као и у Јави, постоји хијерархија наслеђивања типова (слика 2.1). На врху се налази класа *Any* коју свака Скала класа наслеђује, имплицитно

или експлицитно. Овај тип поседује два подтипа, *AnyVal* и *AnyRef*. Први је корен свим Скала типовима који представљају вредности. То су *Byte*, *Short*, *Char*, *Int*, *Long*, *Float*, *Double*, *Boolean* и *Unit*. Друга, *AnyRef*, представља родитељску класу свим референцама у Скали, слично као класа *Object* у Јави. Потомци овог типа се инстанцирају преко кључне речи *new*. [8]

На дну референтних типова се налази класа *Null*. Вредност овог типа се може доделити било којој референци. Једина класа која наслеђује *Null* је *Nothing*. Класа *Nothing* нема вредност и не може се доделити ниједној променљивој.



Слика 2.1: Хијерархија Скала типова

2.8 Скала колекције

Скала поседује велики број уграђених колекција, мутабилних и имутабилних. Неке од њих су низови, листе, скупови и мапе.

Низови

Скала низ (енг. *array*) је мутабилна структура која представља низ података. Мутабилна је у смислу да се вредности елемената у низу могу мењати, док је број елемената фиксан. Сваки низ садржи елементе једног типа и може

се креирати навођењем иницијалних елемената или његове дужине. Уколико се наведе дужина, сви елементи ће бити иницијализовани на подразумевану вредност жељеног типа. [8]

```
scala> val a1 = Array(1, 2, 3)
val a1: Array[Int] = Array(1, 2, 3)

scala> val a2 = new Array[Int](3)
val a2: Array[Int] = Array(0, 0, 0)
```

Кôд 2.36: Инстанцирање низа у Скали

У претходном примеру се у другом случају инстанцира нова класа коришћењем наредбе *new* док у првом то није случај. Разлог је тај што се у првом случају позива метод *apply()* који креира инстанцу низа. Поред тога, у првом случају је Скала компајлер аутоматски закључио тип низа на основу прослеђених елемената, док је у другом тип морао бити експлицитно назначен. [8]

Елементу низа се приступа слично као у Јави, са тим што се уместо угластих заграда користе обичне. На сличан начин се извршава и мењање једног елемента.

```
scala> val a = Array(1, 2, 3)
val a: Array[Int] = Array(1, 2, 3)

scala> a(0)
val res0: Int = 1

scala> a(0) = 100

scala> a
val res1: Array[Int] = Array(100, 2, 3)
```

Кôд 2.37: Приступ и измена елемента низа

Листе

Скала листе представљају имутабилну колекцију елемената истог типа. Разлика листе у Скали у односу на Јавину је та што је Скала листа увек

имутабилна, док Јава листа може бити мутабилна.

Инстанцира се навођењем елемената. Приступ елементу листе се извршава на исти начин као и у случају низа. Пошто су листе имутабилне, измена вредности елемената није дозвољена. [8]

```
scala> val l = List("a", "b", "c")
val l: List[String] = List(a, b, c)

scala> l(0)
val res0: String = a

scala> l(0) = "try"
       ^
       error: value update is not a member of List[String]
       did you mean updated?
```

Кôд 2.38: Пример Скала листе

Спајање листи се извршава оператором :::. Приликом позива овог оператора се не извршава додавање елемената једне листе на другу, већ је резултат нова листа. Овај метод се може користити и за спајање више од две листе. [8]

```
scala> val l1 = List(1, 2, 3)
val l1: List[Int] = List(1, 2, 3)

scala> val l2 = List(4, 5, 6)
val l2: List[Int] = List(4, 5, 6)

scala> val l3 = List(7, 8, 9)
val l3: List[Int] = List(7, 8, 9)

scala> l1 ::: l2 ::: l3
val res0: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

Кôд 2.39: Спајање листи

Нове листе се могу инстанцирати и коришћењем оператора ::, који као аргументе прима елемент и листу истог типа од којих конструише нову листу где елемент додаје на почетак листе.

```
scala> val l = List(1, 2, 3)
val l: List[Int] = List(1, 2, 3)

scala> 10 :: l
val res0: List[Int] = List(10, 1, 2, 3)
```

Кôд 2.40: Пример оператора ::

Коришћењем овог оператора се може извршити надовезивање елемената на празну листу (пример 2.41). У Скали се празна листа означава кључном речи *Nil*.

```
scala> val l = 1 :: 2 :: 3 :: 4 :: Nil
val l: List[Int] = List(1, 2, 3, 4)
```

Кôд 2.41: Додавање елемената на празну листу

Н-торке

Имутабилна колекција која садржи елементе различитог типа се назива н-торка (енг. *Tuple*). Ова структура података се може користити када је потребно вратити више различитих вредности функције. *Tuple* се инстанцира навођењем елемената између заграда. Елементу се приступа оператором _X где је X редни број елемента унутар н-торке. [8]

```
scala> val t = (1, "string123", Array(1, 2, 3))
val t: (Int, String, Array[Int]) = (1,string123,Array(1, 2,
    3))

scala> t._1
val res0: Int = 1

scala> t._3
val res1: Array[Int] = Array(1, 2, 3)
```

Кôд 2.42: Н-торка у Скали

Скупови

Скуп је колекција за коју важи да садржи елементе истог типа, са тим што сваки елемент колекције мора бити јединствен. Постоје две врсте скупова, имутабилни (*collection.immutable.Set*) и мутабилни (*collection.mutable.Set*).

Инстанцирају се на исти начин као и низ, навођењем елемената. Уколико се током навођења наведе неки елемент више од једног пута, не добија се грешка, већ се дупликат аутоматски уклања. Елемент се додаје оператором +. Код мутабилних скупова + додаје елемент на постојећи скуп, док код имутабилних производи нови скуп са додатим елементом. [8]

```
scala> val s = Set(1, 1, 1, 1, 2, 2)
val s: scala.collection.immutable.Set[Int] = Set(1, 2)
```

Кôд 2.43: Инстанцирање скупа у Скали

Мапе

Мапе су колекције за рад са кључ-вредност паровима. Постоје мутабилне (*collection.mutable.Map*) и имутабилне (*collection.immutable.Map*). Имутабилне су подразумеване и користе се уколико се експлицитно не наведе супротно.

Дефинишу се навођењем низа кључ-вредност парова, раздвојених знаком − >. Сви кључеви и све вредности међусобно морају бити истог типа. Приступ вредностима се врши преко назива кључа, методом ().

```
scala> val m1 = Map("k1" -> "v1", "k2" -> "v2")
val m1: scala.collection.Map[String,String] = Map(k1 -> v1,
    k2 -> v2)

scala> m1("k1")
val res0: String = v1

scala> val m2 = Map(1 -> Array(1, 2), 2 -> Array(3, 4))
val m2: scala.collection.Map[Int,Array[Int]] = Map(1 ->
   Array(1, 2), 2 -> Array(3, 4))

scala> m2(1)
```

```
val res1: Array[Int] = Array(1, 2)
```

Кôд 2.44: Мапе у Скали

Разлика између мутабилних и имутабилних мапа је та што је код мутабилних могуће изменити број елемената, као и саме елементе, док имутабилна мапа то не подржава.

```
// mutable map
scala> val mutableM = mutable.Map("k1" -> "v1")
val mutableM: scala.collection.mutable.Map[String,String] =
    HashMap(k1 -> v1)

scala> mutableM("k1") = "vred1"

scala> mutableM("k2") = "vred2"

scala> mutableM
val res0: scala.collection.mutable.Map[String,String] =
   HashMap(k1 -> vred1, k2 -> vred2)

// immutable map
scala> val immutableM = Map("k1" -> "v1")
val immutableM: scala.collection.Map[String,String] = Map(
   k1 -> v1)

scala> immutableM("k1") = "vred1"
       ^
       error: value update is not a member of scala.
   collection.Map[String,String]

scala> immutableM("k2") = "vred2"
       ^
       error: value update is not a member of scala.
   collection.Map[String,String]
```

Кôд 2.45: Измена и додавање елемента код мутабилних и имутабилних мапа

2.9 Поклапање образаца

Поклапање образаца (енг. *pattern matching*) је честа карактеристика функционалних језика. Слична је наредби *switch* из Јаве, али нуди више могућности од ње. Састоји се од селектора, који представља израз или променљиву, кључне речи *match* и низа случајева унутар витичастих заграда. Сваки случај се састоји од знака => који се налази између вредности са којом се селектор поклапа и кључне речи *case* са леве стране и израза који ће бити резултат поклапања са десне. [8]

```
selector match {
  case value1 => result1
  case value2 => result2
  case value3 => result3
  ...
}
```

Кôд 2.46: Поклапање образаца у Скали

Разлике између ове наредбе и Јавине наредбе *switch* су:

- *match* наредба увек резултује неком вредношћу
- Када се пронађе одговарајућа вредност, друге вредности након ње се не разматрају. Није потребно користити наредбу *break*
- Уколико ниједна вредност не одговара селектору, појављује се *MatchError*. То значи да увек треба покрити све могуће вредности селектора

Примери коришћења

Поклапање образаца је веома моћан механизам и користи се у великом броју случајева. Један од њих је поклапање константи:

```
scala> def describe(x: Any) = x match {
     | case 5 => "five"
     | case true => "truth"
     | case "hello" => "hi!"
     | case Nil => "the empty list"
     | case _ => "something else"
```

```
     | }

scala> describe(5)
val res1: String = five

scala> describe(Nil)
val res2: String = the empty list

scala> describe(1001)
val res3: String = something else
```

Кôд 2.47: Поклапање константи

У претходном примеру се у последњем случају користи ознака _. Назива се џокер (енг. *wildcard*) и поклапа се са било којом вредношћу селектора.

```
scala> def describe(x: Int) = x match {
     | case 10 => "x je 10"
     | case _ => "x nije 10"
     | }
def describe(x: Int): String

scala> describe(10)
val res1: String = x je 10

scala> describe(100)
val res2: String = x nije 10

scala> describe(-100)
val res3: String = x nije 10
```

Кôд 2.48: Џокер у поклапању образаца

Поклапање образаца се користи и за поклапање променљивих. У примеру 2.49 променљива *something* одговара било којој вредности селектора, слично као џокер, али се преко ње та вредност преноси са десне стране ознаке =>. [8]

```
scala> val expr = "some expression"
val expr: String = some expression
```

```
scala> expr match {
     | case ""          => print("empty string")
     | case something => print("matched: " + something)
     | }
matched: some expression
```

Кôд 2.49: Поклапање променљивих

Поред вредности селектора, могуће је поклапати и његов тип, уколико је потребно имплементирати другачије понашање за другачије типове .

```
scala> trait MyTrait
trait MyTrait

scala> class MyClass_1 extends MyTrait
class MyClass_1

scala> class MyClass_2 extends MyTrait
class MyClass_2

scala> def describe(x: MyTrait) = x match {
     |   case mc1: MyClass_1 => "MyClass_1"
     |   case mc2: MyClass_2 => "MyClass_2"
     |   case _ => "some other class"
     | }
def describe(x: MyTrait): String

scala> describe(new MyClass_1)
val res1: String = MyClass_1

scala> describe(new MyClass_2)
val res2: String = MyClass_2
```

Кôд 2.50: Поклапање типова

Могуће је комбиновати претходне примере и извршити поклапања типова и променљивих уз помоћ једног поклапања образаца. Пример 2.51 приказује

функцију *generalSize(x: Any)* која се понаша другачије у односу на то ког типа је њен аргумент. [8]

```
scala> def generalSize(x: Any) = x match {
     |   case s: String => s.length
     |   case m: Map[_, _] => m.size
     |   case _ => -1
     | }
def generalSize(x: Any): Int

scala> generalSize("some string")
val res1: Int = 11

scala> generalSize(12)
val res2: Int = -1
```

Кôд 2.51: Пример поклапања типова и променљивих

Поклапање конструктора се користи када је потребно извршити поклапање класа које наслеђују заједничку класу.

```
scala> abstract class Animal
     | case class Mammal(name: String) extends Animal
     | case class Bird(name: String) extends Animal
class Animal
class Mammal
class Bird

scala> def caseClassesPatternMatching(animal: Animal) =
   animal match {
     |   case Mammal(name) => s"I'm a $name, a mammal"
     |   case Bird(name) => s"I'm a $name, a bird"
     |   case _ => "I'm an unknown animal"
     |}
def caseClassesPatternMatching(animal: Animal): String
```

Кôд 2.52: Поклапање конструктора

Глава 3

Дистрибуирана обрада података

У последњих неколико година се генерише огромна количина података. Друштвене мреже, видео садржај и куповина преко интернета на **IoT?** дневном нивоу производе петабајте података, а у будућности се очекује пораст тог тренда. [7]

Све те податке је потребно искористити на користан начин, те је стога питање обраде и анализе, али и складиштења, веома битно. Како је често проблем обрадити петабајте података на једној машини, индустријски стандард су постали кластери, који раде се подацима на дистрибуиран начин.

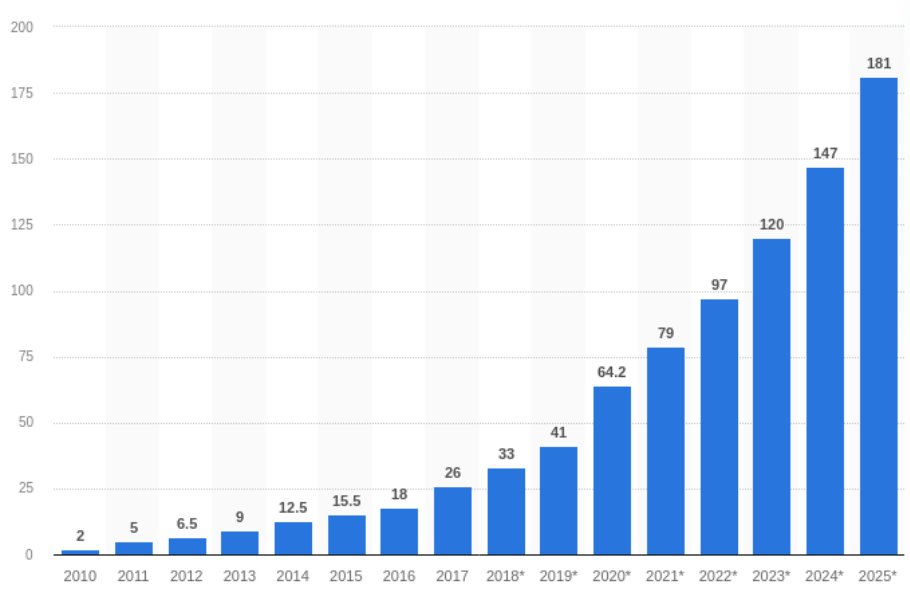
3.1 Доба података

Према истраживању приказаном на једном од водећих интернет платформи за податке који се користе у пословању, *Statista*, количина података која се производи се тренутно може мерити у десетинама зетабајта (енг. *zetabyte*) [1]. Исто истраживање приказује да ће се наредних година тај број удвостручити. Приказ пораста тренда генерисања података је приказан на слици 3.1. [7]

Корист података је огромна и велики број индустрија и компанија их користи на разне начине, од побољшања искуства корисника који користе њихове услуге, до разних предвиђања у пословању. Из тих разлога се доста улаже у складиштење, обраду, истраживање и анализу података.

Раније су подаци, док још увек нису генерисани у количинама у којима се то дешава данас, обрађивани на појединачним машинама, али се убрзо

[1] Милијарда терабајта



Слика 3.1: Количина података по години у зетабајтима

испоставило да такав приступ има своја ограничења.

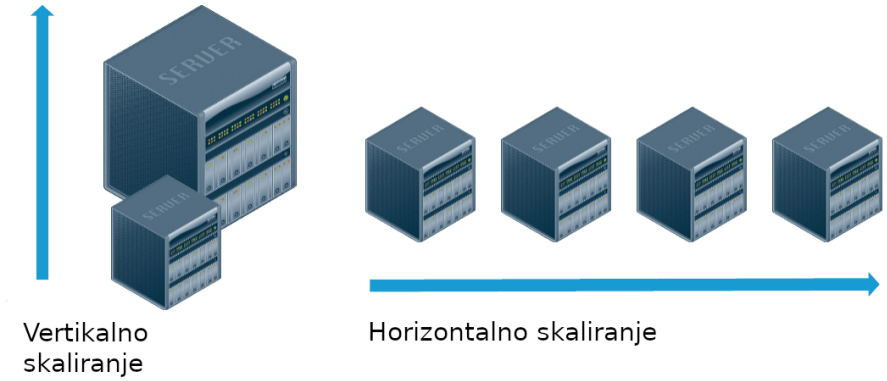
3.2 Скалирање система

У контексту система, скалирање означава могућност система да се прилагоди количини података који се уз помоћ њега обрађују. Постоје два начина скалирања уређаја који врше обраду података (слика 3.2). Први начин је вертикално скалирање (енг. *vertical scaling* или *scale-up*). У овом приступу се унапређује једна машина, на пример, додаје јој се више меморије или се појачава снага процесора. Предност овог приступа је та што се након унапређења машине не мора да мењати логика апликација које се на њој извршавају. Али негативне особине су те што постоји ограничење до ког се машина може унапредити (ипак је потребно обрадити егзабајте података). Такође, пошто је само једна машина у питању, потребно је и обратити посебну пажњу на то шта се дешава ако она доживи некакву грешку и неочекивано престане са радом. [13]

Други приступ је хоризонтално скалирање (енг. *horizontal scaling* или *scale-out*). У овом случају се не унапређује једна машнина, већ се, уколико је потребна додатна снага, додаје нова машина у систем. Добра особина овог приступа је та што је често јефтиније додати неколико нових машина у

систем него унапредити процесор неколико пута на истој машини. Још једна веома добра одлика је ефикасност. Када постоји неколико машина могуће је на свакој од њих обрађивати један део података, што је огромна предност у односу на вертикално скалирање. Међутим, хоризонтално скалирање доноси додатан скуп проблема. Потребно је имплементирати цео систем, омогућити машинама да раде заједно и координисати их, као и обрадити проблеме који се могу десити на појединачним машинама. [13]

Због наведених предности, данашњи стандард у обради података је хоризонтално скалирање.



Слика 3.2: Врсте скалирања система

3.3 Особине дистрибуираних система

Идеја иза хоризонталног скалирања је да свака машина у систему обрађује један део података и да на тај начин доприноси коначном резултату. Одатле следи да машине морају да комуницирају једна са другом и да ће неки подаци можда затребати свакој машини у систему што може довести до такмичења уређаја за приступ тим подацима. Такође, уколико се подаци налазе само на једној машини у систему, све друге машине ће јој приступити, тако да су могућности нашег система у том случају ограничене могућностима те једне машине којој сви приступају. Поред тога, та машина може да искуси некакав проблем и да због тога престане да функционише, што би изазвало престанак рада целог система. [13]

Да би се потенцијални проблеми избегли, систем треба да функционише тако да уређаји који су у њему раде независно од других уређаја истог си-

стема, као и да престанак рада једне машине не утиче на систем у целини. Другим речима, треба направити такав систем који очекује да се фаталне грешке дешавају.

У оваквим системима акценат је на софтверу, а не на хардверу и идеја је да се систем може направити од уређаја који су релативно јефтини и масовно доступни. Такође, циљ је да се избегава премештање података међу уређајима, па се подаци, уколико је то могуће, обрађују на машини на којима се налазе. [13]

3.4 Систем *Hadoop*

Први успешан систем који поседује претходно наведене карактеристике је развила компанија *Google* која је 2003. године објавила научни рад на ту тему [10]. У раду је представљен дистрибуирани фајл систем, назван *Google file system* или скраћено *GFS*. Систем је написан у програмском језику *C++*. Намена овог пројекта је да се користи за складиштење великих количина података. Већ следеће године, *Google* је објавио нови научни рад о парадигми за ефикасну обраду велике количине података на кластеру [2]. Парадигма је названа *MapReduce* и њена намена је да се користи за обраду података складиштених у *GFS*-у.

Недуго након тога, уз помоћ научних радова компаније *Google*, настао је пројекат отвореног кода (енг. *open source*) назван *Hadoop* са идејом да имплементира карактеристике које поседују *Google*-ови *GFS* и *Map Reduce* и да се као такав користи за складиштење и ефикасну обраду падатака на кластеру сачињеном од релативно јефтиних машина. Највећи делови *Hadoop* система су *HDFS* и *MapReduce*, скраћено *MR* који су заправо јавно доступни еквиваленти *Google*-ових технологија. [13]

Укратко, *HDFS* је фајл систем који користи хоризонтално скалирање машина за складиштење огромних количина података. Због боље поузданости користи репликацију, где се сваки фајл копира неколико пута и онда се те копије чувају на различитим уређајима у систему. [13]

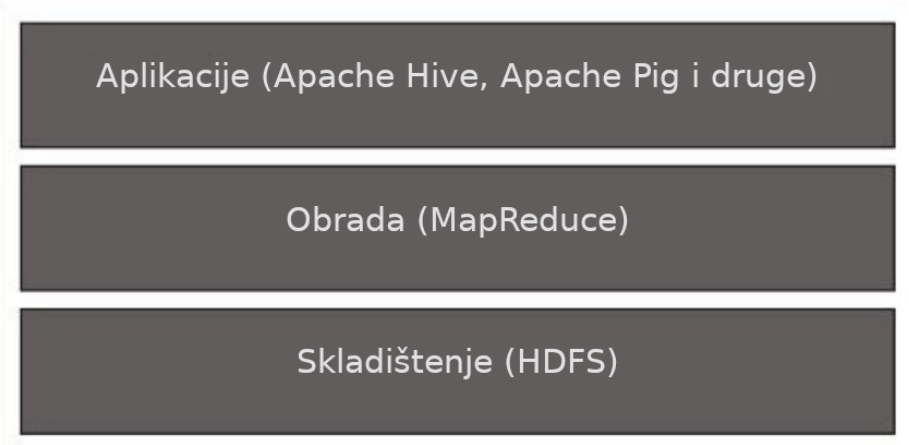
MR је парадигма за обраду података, која се састоји из два дела названа *Map*) и *Reduce*, по којима је парадигма и добила име. У оба дела се над подацима који се налазе унутар *HDFS*-а врши некаква обрада података. Поред тога, сама обрада података се извршава на *HDFS*-у, на истом месту где се они

и налазе. Тиме се избегава премештање података на неку другу машину. [13]

Обе компоненте су направљене да функционишу на релативно јефтином, обичном хардверу. Поред тога користе хоризонтално скалирање и у стању су да наставе са радом чак и ако једна или више машина у систему из неког разлога престане да функционише.

Слика 3.3: Логои *HDFS*-а и *MR*-а

Поред поменуте две компоненте, постоји и трећа, а то су *HDFS* апликације. Оне се надовезују на *HDFS* и *MR* тако што их користе за, редом складиштење и обраду. Најпознатије од њих су *Apache Hive* [3] и *Apache Pig* [4], али поред њих постоје и многе друге, само мање заступљене. На слици 3.4 су приказане компоненте *Hadoop*-а. [14]



Слика 3.4: Упрошћен приказ Хадупа

3.5 Дистрибуирани фајл систем *HDFS*

HDFS, скраћено од *Hadoop distributed file system* је дистрибуирани систем, што значи да складишти податке на више машина које ће се у даљем тексту

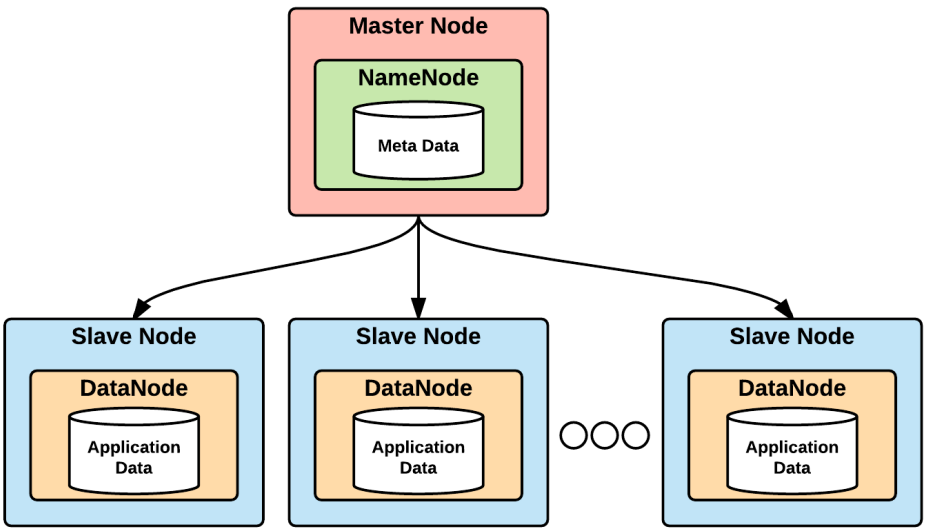
звати чворови (енг. *nodes*). Скуп таквих машина које раде заједно на такав начин да се могу посматрати као једна целина се назива кластер (енг. *cluster*).

Структура *HDFS*-а

Постоје две врсте чворова, именски чвор (енг. *name node*) и чвор података (енг. *data node*). Функционишу по водећи-зависни (енг. *master-slave*) архитектури, где именски чвор има улогу водећег. Чворови су приказани на слици 3.5.

Унутар *HDFS* система се налази један примарни именски чвор чија је улога да управља фајл системом и да регулише приступ подацима који се налазе на њему. Он садржи информације о фајловима, као што су између осталих име, локација у систему где се фајл налази, последњи датум измене фајла као и правила приступа. Поред примарног, *HDFS* може имати и неколико секундарних именских чворова који представљају резервне копије. [5]

Друга врста чворова су чворови података и њихова улога је да складиште фајлове система. Поред тога, на овим чворовима се извршава обрада података. Чворови података су задужени за операције над фајловима као што су читање, мењање и брисање. Они ће извршити неку од тих операција само када им именски чвор то нареди. [5]



Слика 3.5: Врсте чворова у ХДФС-у

Уколико апликација жели да приступи *HDFS*-у, она ће прво комуницирати са именским чвором и од њега затражити фајлове који јој требају. Након тога, именски чвор проверава да ли та апликација поседује потребне дозволе за приступ тим фајловима и ако их она има, послаће јој њихову локацију у фајл систему. Након тога се може извршити жељени приступ.

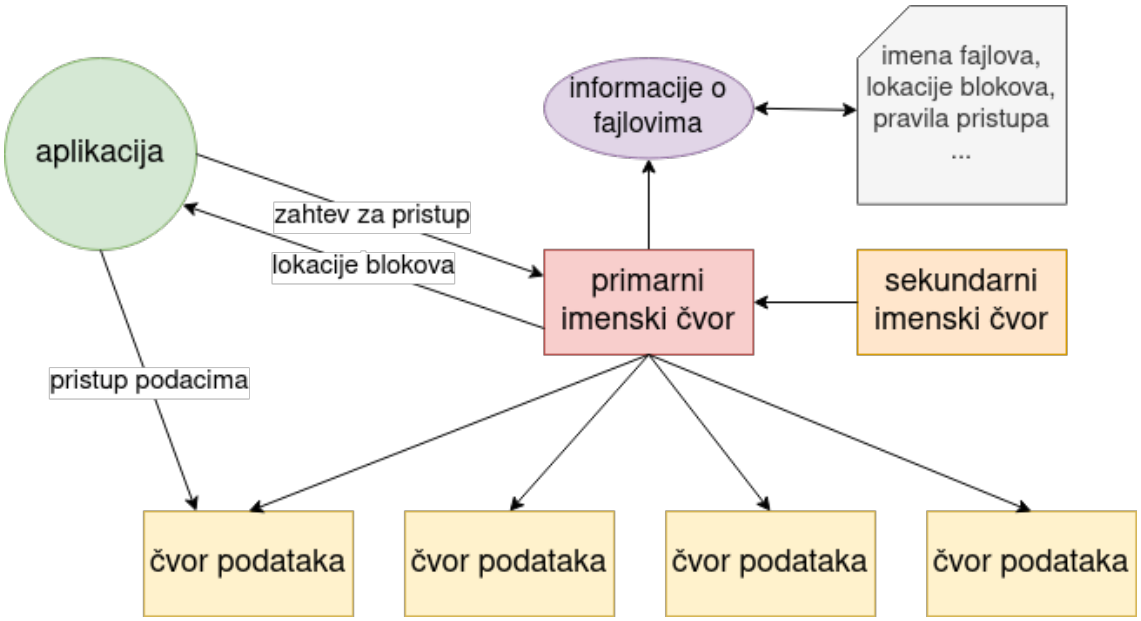
Особине

Сваки фајл у ХДФС-у је подељен на делове који се називају блокови чија је величина обично 128 мегабајта. Блокови се често не налазе на истим чворовима у систему, што значи да се један фајл чува на неколико физички раздвојених машина. Ту може да настане проблем због тога што једна од тих машина може да се поквари и због тога престане са радом. У том сличају, сви блокови складиштени на тој машини ће нестати. Да би се губљење фајлова избегло, *HDFS* сваки блок реплицира неколико пута и након тога оригинални блок и његове реплике распореди по систему. Ако један од блокова фајла неочекивано нестане, увек је могуће приступити једној од његових реплика. Генерисане реплике се чувају на чворовима података, док се информације о томе ком фајлу реплике припадају налазе на именском чвору. [5]

Блок ће се увек реплицирати одређен, фиксиран, број пута. Чворови података повремено шаљу сигнале именском чвору о доступности реплика. На тај начин ће именски чвор увек имати информацију о томе колико је пута сваки блок реплициран у систему и на основу тога може да, уколико тај број падне испод неке задовољавајуће вредности, направи нове реплике тог блока. [5]

HDFS је конструисан тако да може да настави са радом у случају фаталних грешака на чворовима података. Међутим, могућа је појава грешака и на именском чвору и те грешке могу довести до пада целокупног система. Такви проблеми се решавају чувањем резервних копија именског чвора и због њих се у случају престанка његовог рада не губе никакве информације. Резервне копије се праве у одређеним временским интервалима да би подаци на њима били ажурни. Резервне копије и репликација су битне за целокупну робусност система, односно поузданости података чак и у случајевима када се десе некакви проблеми. Концепти *HDFS*-а су приказани на слици 3.6.

HDFS је систем за кога важи *пиши једном, читај више пута* (енг. *write-once, read-many (WORM)*). Када се фајл постави унутар *HDFS*-а више се не



Слика 3.6: Основне *HDFS* компоненте

може мењати. Уколико се фајл мора изменити долази до креирања новог фајла који замењује стари. Иако такав приступ није ефикасан, то често није битно због тога што се апликације које обрађују велике количине података заснивају на томе да се подаци не мењају, па је за очекивати да за променама неће бити потребе или ће такви случајеви бити ретки. [13]

Такође, још једна од особина *HDFS*-а је та да је конструисан да има добре перформансе у случајевима када је потребан велики проток података, на пример у случају читања великих фајлова. [13]

3.6 Парадигма *MapReduce*

MR је парадигма која се користи за обраду података који су складиштени у *HDFS*-у. Користи подели и завладај (енг. *divide and conquer*) приступ приликом обраде података тако да више машина паралелно обрађује један њихов део. На тај начин се, на пример, обрада података која би трајала 1000 минута, паралелизациојом на 1000 машина, могла свести на обраду која траје само један минут. [13]

Парадигма је заснована на концептима функционалног програмирања и функцијама које се често користе у обради низова и листи. Те фунције су *map()* и *reduce()*. Прва од постојеће листе креира нову тако што на сваки

елемент листе примени неку функцију и од њега направи нови елемент. Друга од целе листе производи једну вредност. На истим принципима функционише и *MapReduce* парадигма.

MR обрађује податке у неколико фаза. Прво, подаци се читају из *HDFS*-а и након тога се прослеђују машинама које се зову мапери (енг. *mappers*). Те машине паралелно производе скуп привремених података који се након тога распоређују, сортирају и шаљу машинама које се зову редјусери (енг. *reducers*). Фаза која распоређује податке се назива фаза мешања и сортирања (енг. *shuffle and sort*). Задатак редјусера је да приме подскуп података и да паралелно произведу једну вредност од истих. На самом крају се резултат свих редјусера комбинује и добија се резултат читавог *MR* процеса, другачије званог и *MapReduce* задатак (енг. *task*). Могуће је, уланчавањем, комбиновати *MR* задатке, тако да излаз из једног буде улаз у други. [14]

MapReduce из аспекта функција

Из функционалног аспекта, *map* и *reduce* фазе се могу посматрати на следећи начин. Нек постоји листа почетних података у (кључ, вредност) формату. Због једноставности ће се за поменути формат користити ознака (к, в). Током *map* фазе подаци из листе парова се читају из *HDSF*-а и деле на делове на које се паралелно примењује *map()* функција дефинисана од стране програмера. Паралелизам се постиже тако што се сваки део обрађује на засебној машини, маперу. *Map* фаза као улаз прима листу (к, в) парова и производи нову листу истог формата. [14]

$$(k1, v1) -> map(k1, v1) -> list(k2, v2)$$

Након тога се нове листе генерисане од мапера групишу тако што се за сваки кључ прави једна група која ће бити обрађена од стране једног редјусера. Овај део обраде је *shuffle and sort*.

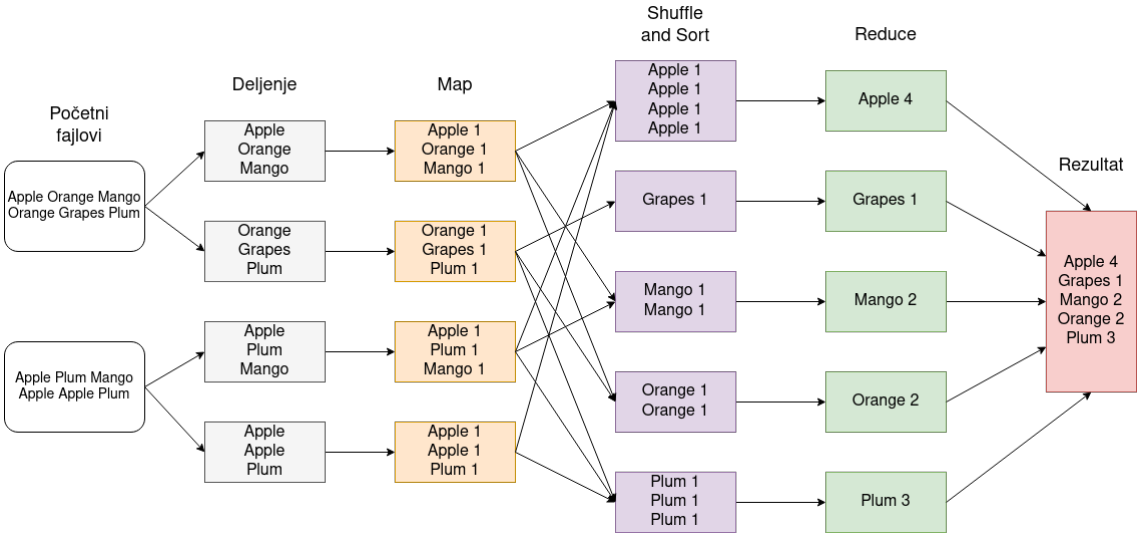
$$list(k2, v2) -> shuffleAndSort(k2, v2) -> k2, list(v2)$$

У последњој фази се на сваку од креираних група примењује *reduce()* функција која производи једну вредност за сваку групу. Овај процес је паралелизован и није могуће да се две групе података са различитим кључевима обрађују на истој машини у истом тренутку. *Reduce* корак прима кључ и

листу вредности које му одговарају и као резултат производи једну вредност формата (к, в). [14]

$$k2, list(v2) -> reduce(k2, list(v2)) -> (k3, v3)$$

Коначан резултат се добија комбиновањем резултата свих редјусера и може се уписати у *HDFS* или се искористити као улаз у другу *MapReduce* апликацију. Цео процес је приказан на слици 3.7 где је као пример представљена *MapReduce* апликација која пребројава број појављивања сваке речи у тексту. У приказаном примеру улаз у мапере је форматиран тако да је редни број линије фајла кључ, док је вредност текст линије. [14]



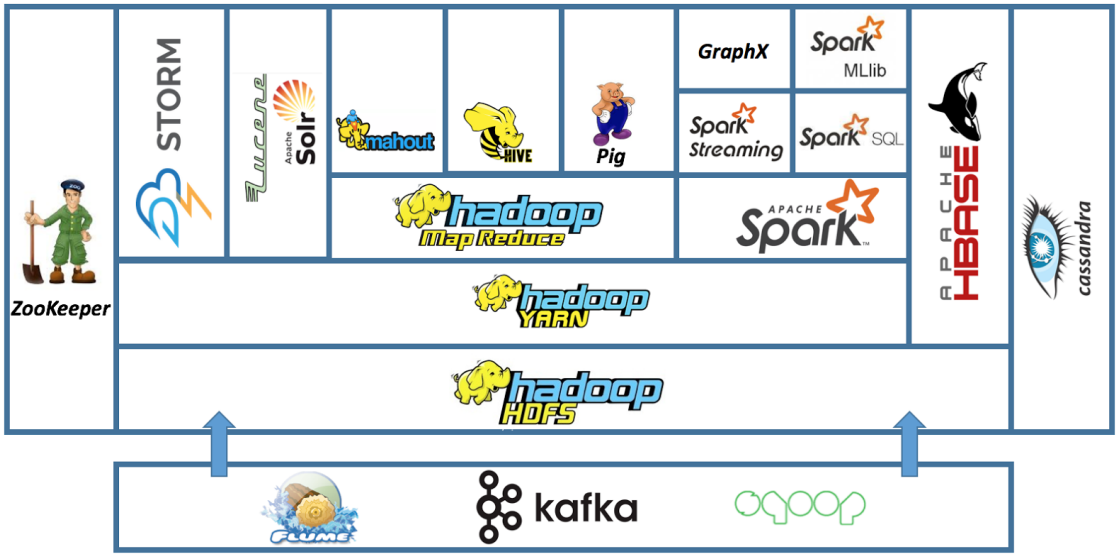
Слика 3.7: Пример *MapReduce* апликације

У *MR* апликацијама задатак програмера је да опише како ће се извршавати *map* и *reduce* фазе, док ће се *Hadoop* систем побринути за све остало: читање података, сортирање, паралелизацију, координацију и извршавање послова. [13]

Ова парадигма је направљена да ради за податке у (кључ, вредност) формату, а не за податке који имају дефинисану шему. Пример података који имају шему би биле табеле у релационим моделима. [13]

3.7 Остале компоненте *Hadoop*-а

Hadoop екосистем чини велики број апликација разних примена које на неки начин користе *HDFS*. Поред самог *HDFS*-а и *MapReduce*-а у њега спадају преговарач ресурса *Apache Yarn*, који ће детаљније бити описан у секцији 3.8, *Apache Kafka*, апликација за рад са токовима података, *Apache Pig* и *Apache Hive*, које се користе за обраду података и имплементиране су користећи *MapReduce*. Поред њих постоје, на пример, *Presto*, *Hue*, *Apache Flume*, *Apache Zookeeper* али и многе друге. Приказ малог дела Хадуп екосистема је приказан на слици 3.8.



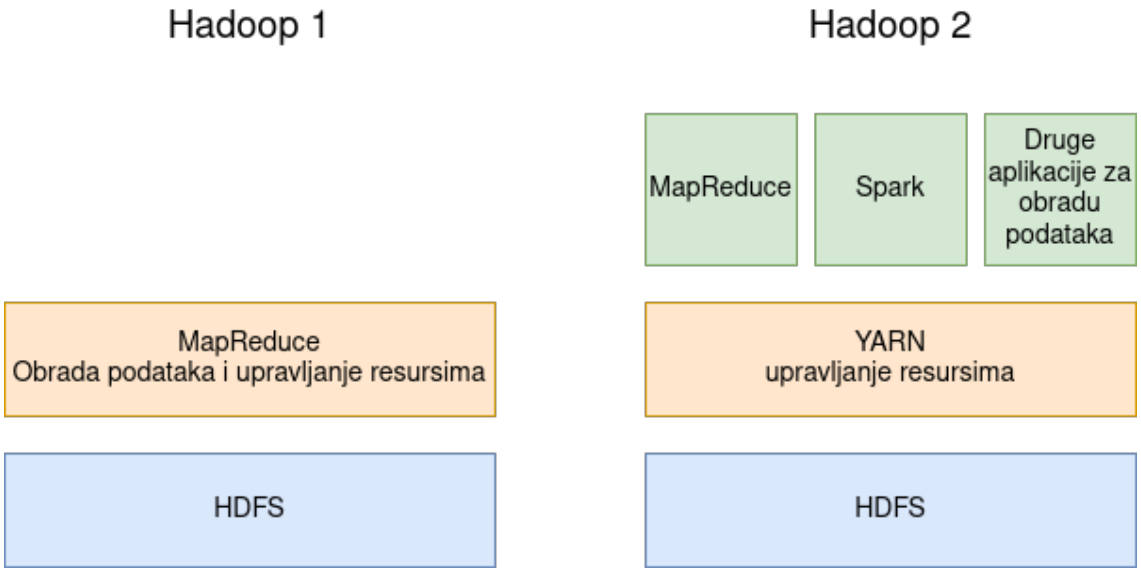
Слика 3.8: Део *Hadoop* екосистема

3.8 Преговарач ресурса *Apache Yarn*

У првој верзији *Hadoop*-а, *MapReduce* је поред обраде великих количина података, за шта је примерно и намењен, имао додатне задатке, а то су алокација и управљање ресурсима који су *MR* апликацији потребни и заказивање *MapReduce* задатака. Таква архитектура је знатно отежавала конструкцију апликација које користе *MR*, па су због тога, у другој верзији *Hadoop*-а, одговорности *MapReduce*-а раздвојене. *MapReduce* је постао алат ексклузивно за

обраду података, док је управљање ресурса предато новој апликацији, коју *MR* током извршавања користи. [14]

Резултат је менаџер ресурса (енг. *resource manager*) отвореног кода назван *Yarn* или *yet another resource negotiator*. Његова улога је да распоређује задатке апликација које користе *Hadoop*, али и да управља ресурсима који су тим апликацијама потребни. Конструисан је да не буде специфичан само за *MapReduce*, већ пружа интерфејс ка *Hadoop*-у разним апликацијама међу којима је и *Apache Spark* (секција 3.9). Разлика у архитектури у различитим верзијама *Hadoop*-а је приказана на слици 3.9. [14]



Слика 3.9: Разлика између *Hadoop* верзија

Архитектура *Yarn*-а

Улога *Yarn*-а, као и других менаџера ресурса, није обрада података, већ је задужен да апликацијама које обрађују податке обезбеди ресурсе потребне за њихово извршавање. Направљен је тако да може да подржи разне врсте апликација. [14]

Конструисан је од две главне компоненте, менаџера ресурса (енг. *resource manager*) и од менаџера чворова (енг. *node manager*). Улога првог је да управља ресурсима читавог кластера, док други управља ресурсима машине на којој је покренут. То значи да ће кластер имати један менаџер ресурса и више менаџера чворова, по један за сваку машину у кластеру. Заједно, они

ће управљати контејнерима (енг. *container*), апстракцијом меморије, процесорске снаге и улазно-излазних операција потребних да би се извршио један део апликације на кластеру. [14]

Менаџер ресурса је најбитнија компонента *Yarn*-а и одговоран је за извршавање сваке апликације на кластеру. Састоји се од две компоненте, заказивача (енг. *scheduler*) и менаџера апликације (енг. *application manager*). Први регулише када се која апликација извршава, док други прихвата апликације којима је кластер потребан и преговара о алокацији првог контејнера који је тим апликацијама потребан. [14]

Улога *Yarn*-а је да обезбеди ресурсе и распореди извршавање задатака на кластеру. Све остало, попут надгледања система, праћења прогреса апликација, обраде грешака и сличног, је имплементирано у коду апликације која га користи. Идеја иза тога је да *Yarn* буде што је више могуће самосталан, да би различите врсте апликација могле да га користе. [14]

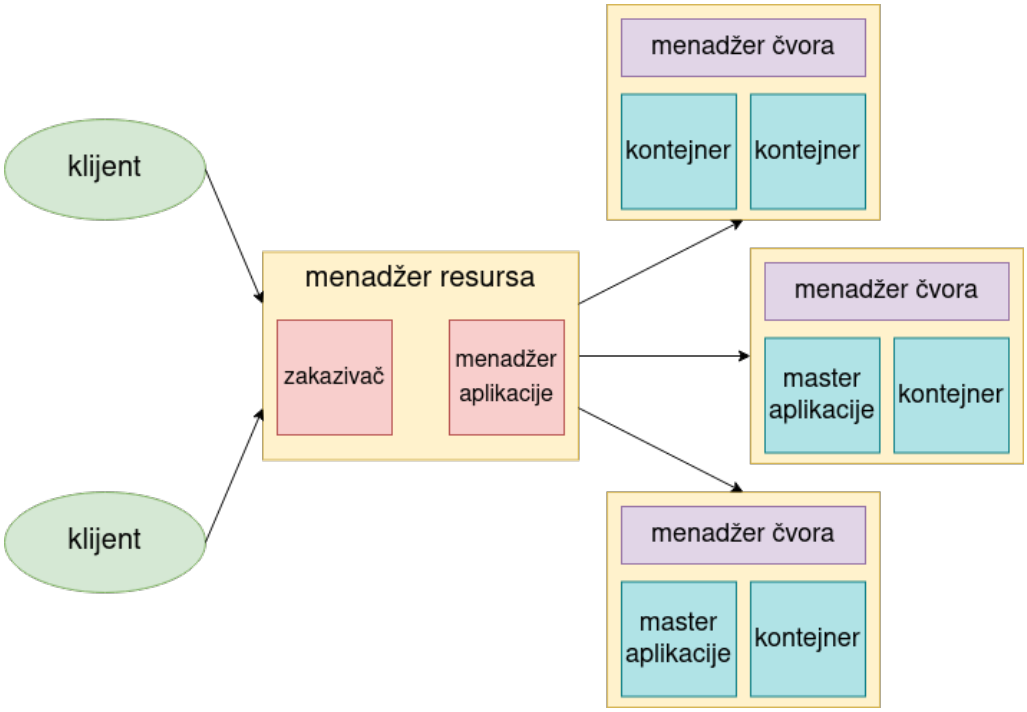
Апликација која се покреће преко *Yarn*-а се састоји из два дела. Први је кôд који треба извршити на кластеру, док се други зове мастер апликације (енг. *application master*). Његова улога је да преговара о ресурсима и прати прогрес и статус апликације. *Yarn* нема информацију о томе на који начин је успостављена комуникација између мастера апликације и кода који се извршава. Приказ архитектуре и компоненти *Yarn*-а је приказан на слици 3.10. [14]

Процес покретања апликације преко *Yarn*-а се извршава следећим редоследом:

1. Клијент пријављује апликацију
2. Менаџер ресурса алоцира контејнер на чвору у коме се покреће мастер апликације
3. Мастер апликације се региструје код менаџера ресурса
4. Мастер апликације преговара о контејнерима са менаџером ресурса
5. Мастер апликације комунуцира са менаџером чвора о покретању потребних контејнера за извршавање апликације
6. Кôд апликације се извршава унутар контејнера

7. Клијент преко менаџера ресурса и мастера апликације прати прогрес апликације

8. Процес је завршен, мастер апликације се одјављује од менаџера ресурса



Слика 3.10: Архитектура *Yarn*-а

Yarn има одговорност да омогући правилно извршавање апликацијама које се извршавају на *HDFS*-у па стога мора обрадити грешке које се могу појавити. На пример, могуће је да једна од машина у кластеру престане се радом и тако постане неупотребљива. Када се то деси, менаџер ресурса ће менаџер чвора на тој машини означити мртвим и неће га више разматрати. Исто ће се десити и са контејнерима те машине. Такође, сваки контејнер који почне да користи више ресурса од оних који су му омогућени ће бити уништен, да не би изазивао проблеме другим апликацијама у систему. [14]

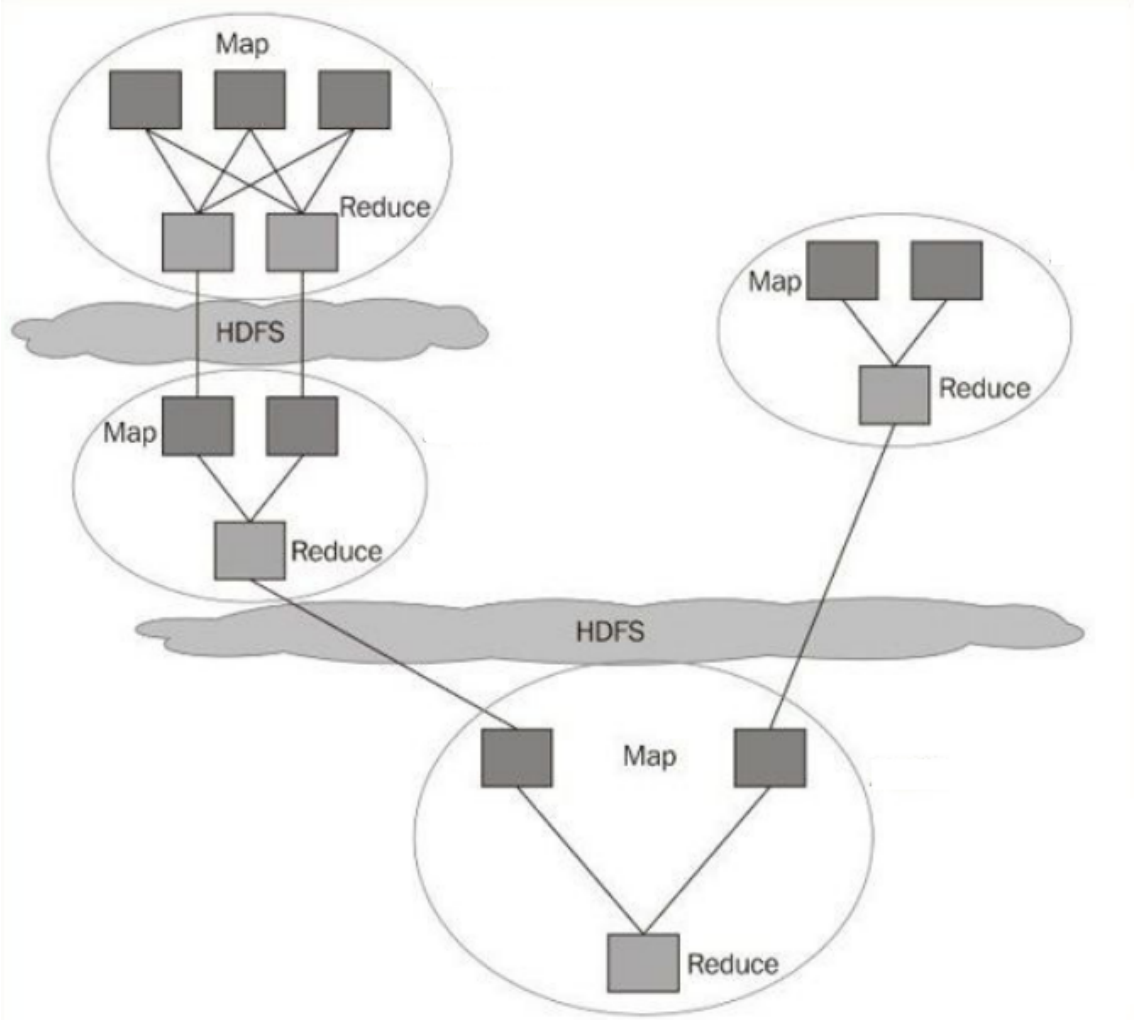
3.9 Алат *Apache Spark*

Иако је *MapReduce* парадигма обележила почетак доба обраде великих количина података, у данашње време се ретко користи. Примаран разлог томе

су недовољно добре перформансе. Потиснута је од стране других технологија и алата, међу којима је и *Apache Spark*.

Недостаци *MapReduce* парадигме

Једна *MapReduce* апликација се састоји од ланца *MapReduce* послова, таквих да излаз једног посла представља улаз у други. Сваки од њих садржи *map* и *reduce* фазу. Резултат апликације представља излаз генерисан од стране последњег посла у ланцу. Пример *MapReduce* апликације је приказан на слици 3.11.



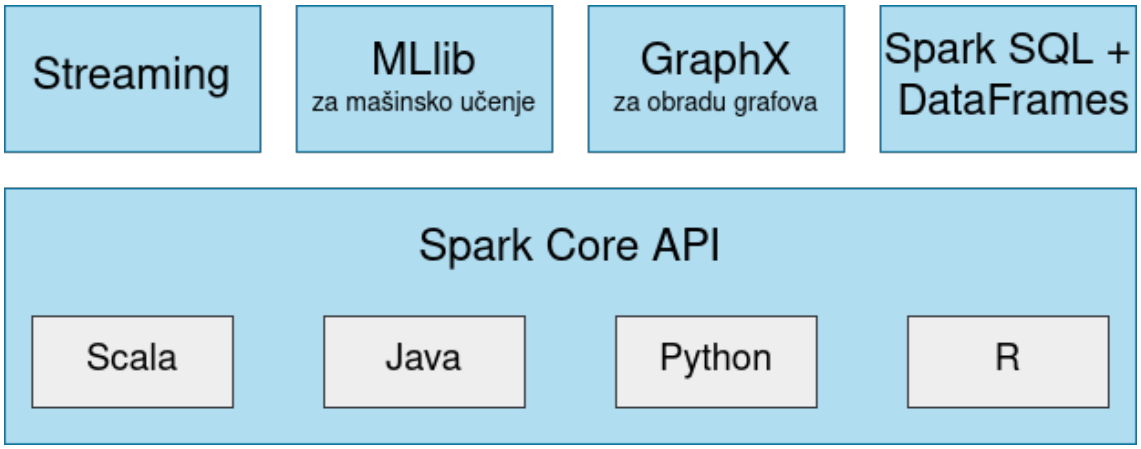
Слика 3.11: пример ланца *MapReduce* послова

Овакав приступ има цену, а то је да се излаз генерисан од стране једног

MapReduce посла чува унутар *HDFS*-а, одакле му приступају други *MapReduce* послови којима је тај излаз потребан. Другим речима, међурезултати послова се чувају на диску, што ствара додатне input/output операције и тиме успорава извршавање целокупне апликације. Поред тога, унутар *MapReduce* парадигме, не постоји аутоматски начин да се послови заједно оптимизују, на пример комбиновањем, већ је за то задужен програмер. Да би се поменути проблеми решили, настао је велики број технологија међу којима је и *Apache Spark*. [14]

Увод у *Apache Spark*

Apache Spark је алат отвореног кода намењен за дистрибуирану обраду велике количине података. Поред тога се користи за рад са токовима података (енг. *streaming*), машинско учење и рад са графовима. Подржан је у програмским језицима *Scala*, *Java*, *Python* и *R*. Иако је намењен за рад на кластерима, лако се скалира па се може користити и на једној машини. На слици 3.12 су приказане компоненте *Apache Spark*-а. У овој секцији ће детаљније бити приказане оне за обраду података. [1]



Слика 3.12: Компоненте *Apache Spark*-а

Настао је 2009. године на универзитету Беркли (енг. *Berkeley*) у Калифорнији. Написан је у програмском језику Скала и конструисан је са идејом да користи концепте функционалног програмирања. Постао је део *Apache* фондације 2013. године. Од тада су избачене три верзије, редом назване, *Spark* 1.0 (2013. године), *Spark* 2.0 (2016. године) и *Spark* 3.0 (2020. године).

Архитектура

Да би *Spark* могао да приступа кластеру, потребно му је омогућити приступ уз помоћ ресурс менаџера. Иако *Spark* има сопствени менаџер ресурса, могу се користити и други, попут *Apache Yarn*-а. Након повезивања је могуће покренути *Spark* апликације на кластеру.

Свака *Spark* апликација се састоји из једног драјвер процеса (енг. *driver process*) и једног или више егзекјутор процеса (енг. *executor process*). Драјвер процес је срце *Spark* апликације и има три задужења:

- прикупљање информација о апликацији која се извршава
- реаговање на програм апликације и њен унос
- анализирање, распоређивање и планирање послова који се извршавају на егзекјуторима

Егзекјутори имају улогу да извршавају посао који им драјвер процес задаје. Поред тога, задужени су и за пријављивање стања извршавања тог посла драјверу.

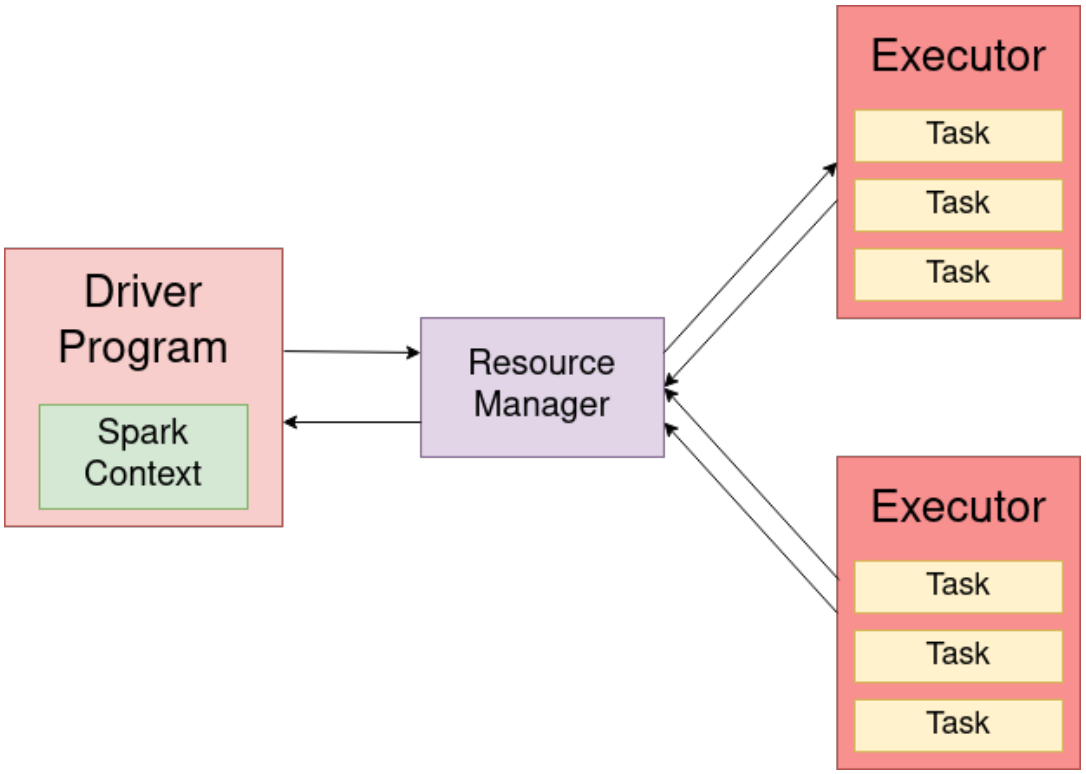
Унутар драјвера постоји одвојен процес назван спарк контекст (енг. *spark context*). Његова улога је да дефинише конекцију ка кластеру. Такође се користи и за креирање апстракција *Apache Spark*-а названих *RDD* (поглавље 3.9). Једноставан приказ апрхитектуре *Spark*-а је приказан на слици 3.13. [1]

Архитектура је заснована на истим концептима, независно од тога да ли се *Spark* покреће у локалном моду, на једној машини, или на кластеру. Једина разлика је у томе што се на кластеру драјвер и егзекјутори налазе на различитим машинама, док ће локално бити покренути на истој. [1]

Партиције

Да би егзекјутори могли паралелно да извршавају операције над подацима, *Spark* их дели на делове који се називају партиције (енг. *partitions*). Партиција је један део колекције података који се налази на једној машини кластера. Идеја је да се операције над партицијама у исто време извршавају на различитим машинама, чиме се постиже паралелизам. [1]

Уколико су подаци партиционисани само једном партицијом, биће обрађени од стране једне машине у кластеру, независно од тога колико машина



Слика 3.13: Архитектура *Apache Spark*-а

постоји. Слично, уколико је креирано више партиција, али постоји само један егзекјутор, ниво паралелизма ће бити један због тога што постоји само једна машина која може да обради те податке. [1]

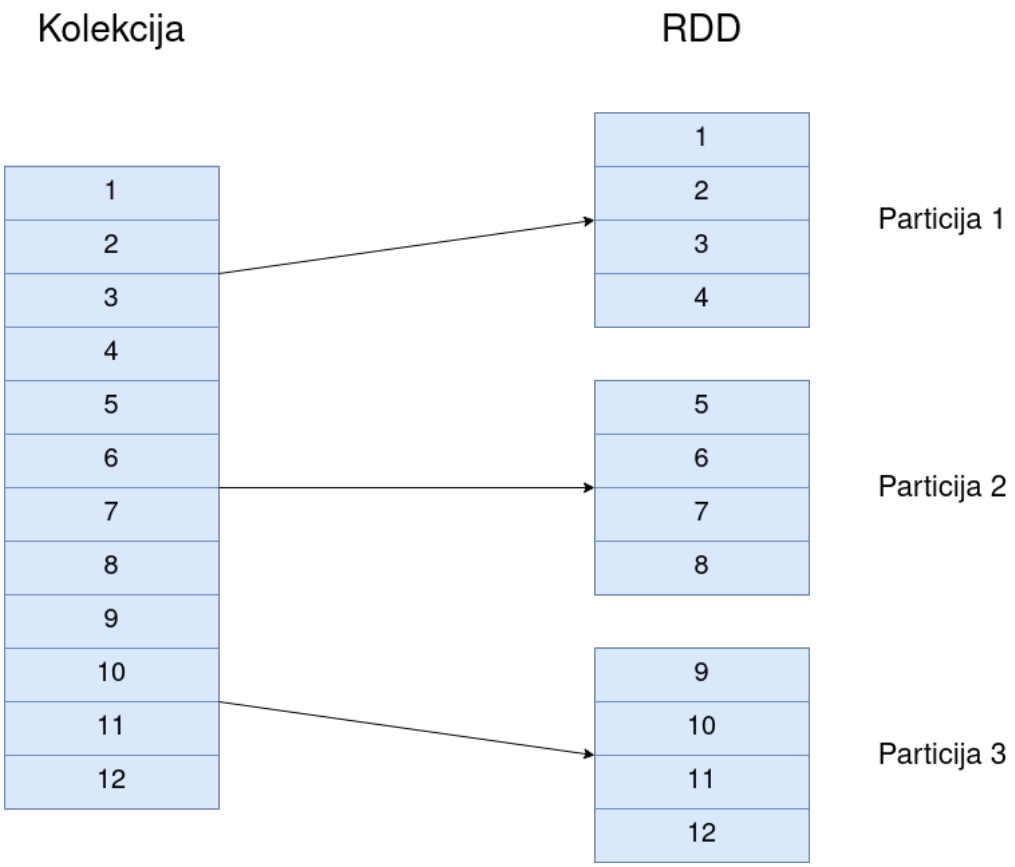
RDD апстракција

Основна јединица рада у *Spark*-у се назива *RDD* (енг. *resilient distributed dataset*) и све операције са подацима се извршавају преко ње. *RDD* је колекција елемената за које важи да су партиционисани по машинама кластера и да се над њима паралелно могу извршавати операције. [11].

Постоји неколико начина преко којих се *RDD* може креирати:

- Читањем неког фајла који се налази на фајл систему (обично *HDFS*)
- Од колекција података програмског језика у коме се користи *Spark* (слика 3.14). Процес дељења колекције у партиције се назива паралелизација
- Од већ постојећег *RDD*-ја

- Кеширањем постојећег *RDD*-ја



Слика 3.14: Креирање *RDD*-ја од колекције података

Данас се *RDD* апстракција не користи директно, већ постоје друге које су конструисане над њом и које су је потиснуле, углавном због бољих перформанси, попут *Spark DataFrame*-а (поглавље 3.9). Због тога се *RDD* сматра застарелим.

Трансформације

Spark је конструисан по принципима функционалног програмирања, па су све његове структуре података имутабилне, што значи да се након њиховог креирања не могу мењати. Пошто се подаци не могу мењати, свака операција која треба да их измени заправо креира потпуно нову струкуру података. На пример, уколико постоји *RDD* којем се мењају подаци које садржи, неће се

изменити директно, већ ће се од њега направити нови *RDD* који у себи саржи измењене податке. [1]

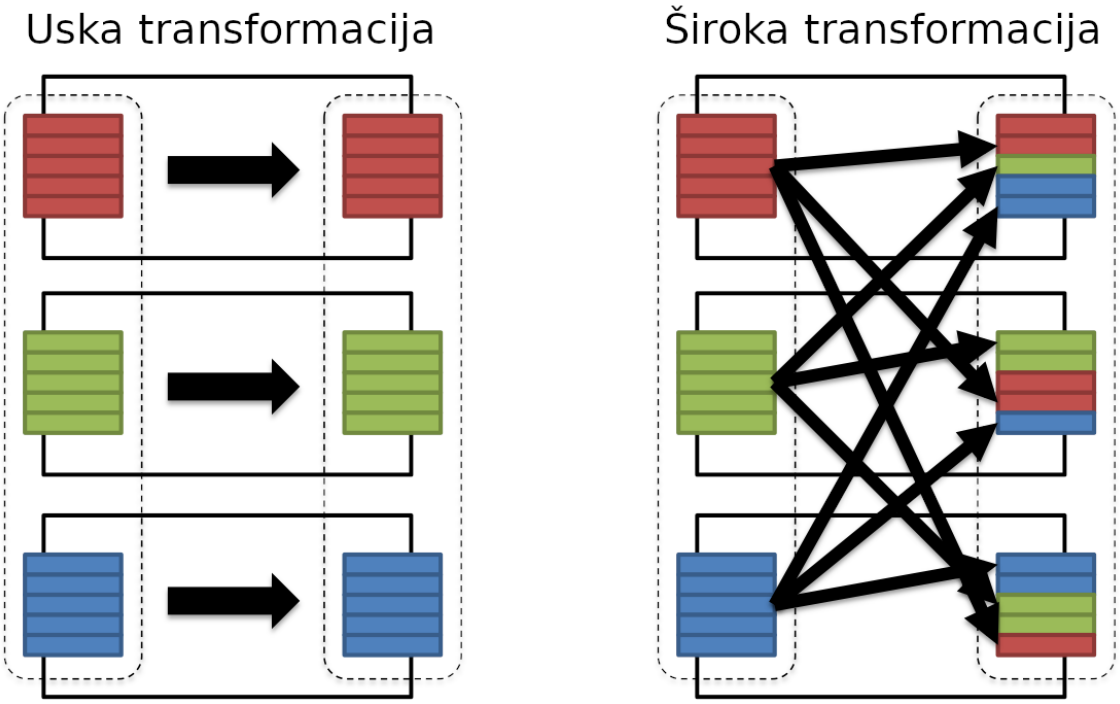
Тај процес, где се од једног *RDD*-ја применом неких наредби добија други, се назива трансформација (енг. *transformation*). Пратећи функционалне концепте, трансформације немају бочне ефекте, што значи да се од једног *RDD*-ја применом истих трансформација, као резултат увек добија исти *RDD*, независно од тога када се те трансформације примењују. *RDD* који трансформацијом настаје од другог *RDD*-ја се назива зависни *RDD* (енг. *dependency*). [1]

Постоје две различите врсте трансформација, уске (енг. *narrow*) и широке (енг. *wide*). За уске трансформације важи да једна партиција у оригиналном *RDD*-ју доприноси настајању највише једне партиције у зависном *RDD*-ју. Са друге стране, широке трансформације су такве где једна партиција почетног *RDD*-ја учествује у конструисању више партиција зависног *RDD*-ја, често свакој. Врсте трансформација су приказане на слици 3.15. Из слике се за широку трансформацију може приметити да се подаци унутар једне партиције изворног *RDD*-ја премештају у сваку партицију зависног *RDD*-ја. Та појава се другачије назива мешање (енг. *shuffle*). [1]

Постоји значајна разлика у перформансама узмеђу уских и широких трансформација. Код уских, *Spark* извршава операције у меморији, док код широких пише резултате на диск и поново их распоређује по партицијама, што значајно успорава извршавање. [1]

Лења евалуација

Лења евалуација је начин евалуације у коме се израз евалуира тек када се његова вредност затражи. *Spark* трансформације припадају лењој евалуацији, па ће се ланац трансформација над неким подацима извршити тек онда када је њихов резултат потребан. Ако постоји *RDD* над којим је дефинисано неколико трансформација, оне неће бити одмах извршене, већ ће се од њих конструисати план трансформација који ће се извршити тек када се цео ланац евалуира. Разлог оваквог приступа је ефикасност. Када *Spark* зна које ће се трансформације извршити над неком структуром података, може оптимизовати цео процес на такав начин да добијање резултата траје најкраће могуће. [1]



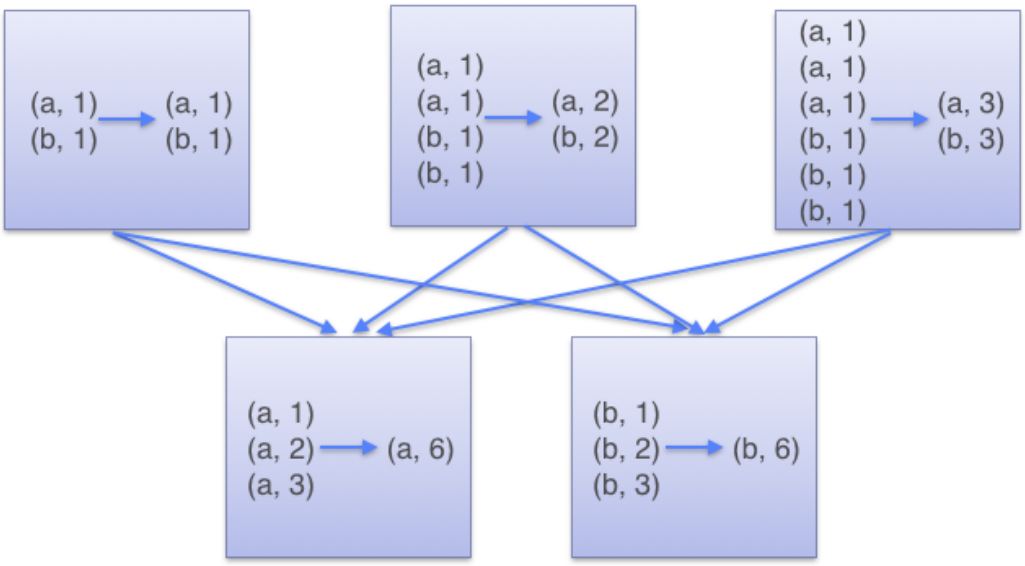
Слика 3.15: Приказ трансформација *Spark*-а

Примери *RDD* трансформација

У *Spark*-у постоји велики број трансформације. Вероватно најпознатија је *map* трансформација која за сваки елемент почетног скупа података производи нови, примењујући неку операцију на њега. Поред класичне *map* трансформације постоји и *flatMap*, која за сваки елемент производи нула, један или више елемената. Још једна трансформација која се често користи је *filter*. Она од постојећег скупа елемената производи нови у коме се налазе они елементи почетног скупа који задовољавају некакав услов. Све три напомените трансформације су уске.

Од широких трансформација се често користе *reduceByKey*, редуковање по кључу, и *join*. Обе раде са кључ-вредност паровима. Прва сабира вредности које имају заједнички кључ (слика 3.16). *Join*, на основу кључева, спаја два скупа елемената у један, где ће резултат бити скуп података у коме је вредност сваког кључа унија вредности тог кључа у засебним скупима који учествују у спајању.

Поред ових трансформација постоје и *union*, *intersect*, *mapValues*, *sortByKey*, *groupByKey* али и многе друге. [11]



Слика 3.16: Редуковање по кључу

Акције

Spark акције се користе када је потребно добити резултат ланца трансформација. Нпомена да извршавање трансформација почиње тек када се њихов резултат затражи, дакле када се позове акција. Добијени резултат се враћа драјвер програму. [1]

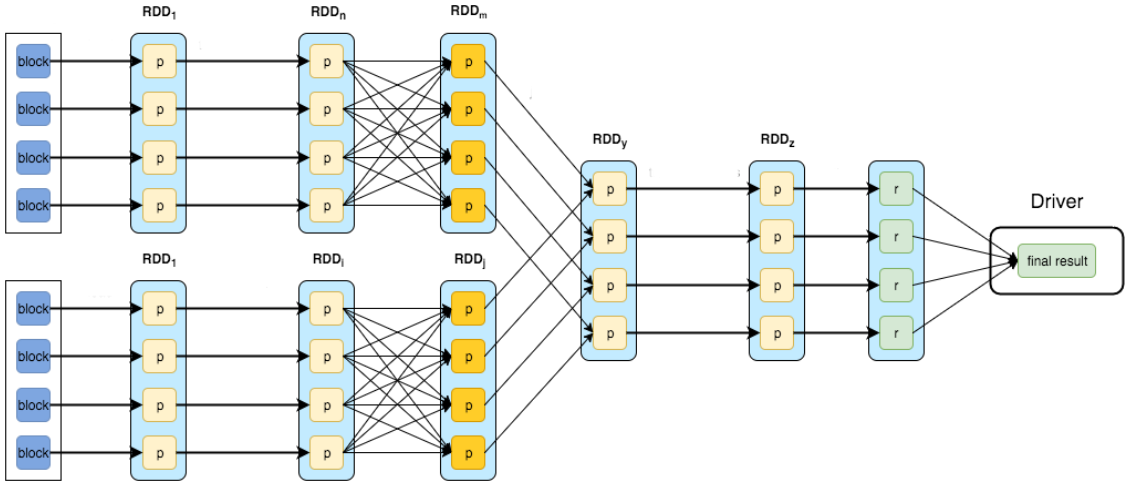
Постоје три врсте акција:

- акције које приказују резултат у конзоли
- акције које исписују резултат у излаз, на пример фајл
- акције које пребацују податке у објекте неког програмског језика у коме се користи *Spark*

Неке од акција су *count* која исписује број елемената у структури података, *saveAsTextFile* која чува податке у документ као и *take* и *collect*, које пребацују податке из *Spark*-а у колекције програмског језика. Поред њих постоје и многе друге. [11]

Руковање грешкама

Могуће је да се током извршавања догоде неки проблеми и да се због тога изгубе партиције. Уколико се то деси, *Spark* је у могућности да реконструише изгубљене партиције уз помоћ механизма који се назива граф наследства (енг. *lineage graph*). За сваки *RDD*, *Spark* чува информације о томе од ког је *RDD*-ја настао и које су трансформације примењене да би се добио тај *RDD*. Пример једног ланца *Spark* трансформација који у исто време представља и граф наследства је приказан на слици 3.17.



Слика 3.17: Пример једног *Spark* извршавања

Уз помоћ овог механизма *Spark* зна од које партиције је настала свака партиција и уколико нека од њих нестане, може је рекреирати. Рекреирање се извршава проласком кроз граф наследства. Ако партиција није валидна, *Spark* ће проверити све партиције од којих је она настала. Уколико оне постоје, рекреираће невалидну партицију од њих, примењујући потребне трансформације. У супротном, прегледаће рекурзивно изворне партиције тих партиција. И то ће радити све док не пронађе валидну партицију, или не дође до партиције која је настала директним читањем из меморије. У том случају ће је прочитати поново и покренути цео ланац трансформација из почетка. [14]

Процес рекреације је поуздан из два разлога. Први је тај што трансформације немају бочне ефекте, па ће се рекреирањем увек добити један исти *RDD*. Други је тај што се подаци чувају у *HDFS*-у, који је сам по себи поуздан, па ће се у случају поновног читања из меморије и рекреирања читавог ланца, увек прочитати непромењена, почетна, вредност са диска.

Кеширање

Веома битна карактеристика *Spark*-а је могућност чувања података у меморији, односно кеширања. Када се *RDD* кешира, свака машина у кластеру ће у својој меморији сачувати партиције које се на њој налазе и касније ће их користити у акцијама или трансформацијама у којима је тај *RDD* потребан, без извршавања целог ланца трансформација из почетка. Овакав приступ знатно побољшава перформансе *Spark* апликације. [11]

RDD се може кеширати коришћењем *cache()* или *persist()* функција. Кеширан *RDD* се чува у меморији тек након што учествује у некој акцији. Кеширање је отпорно на грешке, и за рекреацију несталих партиција се користи граф наследства. [11]

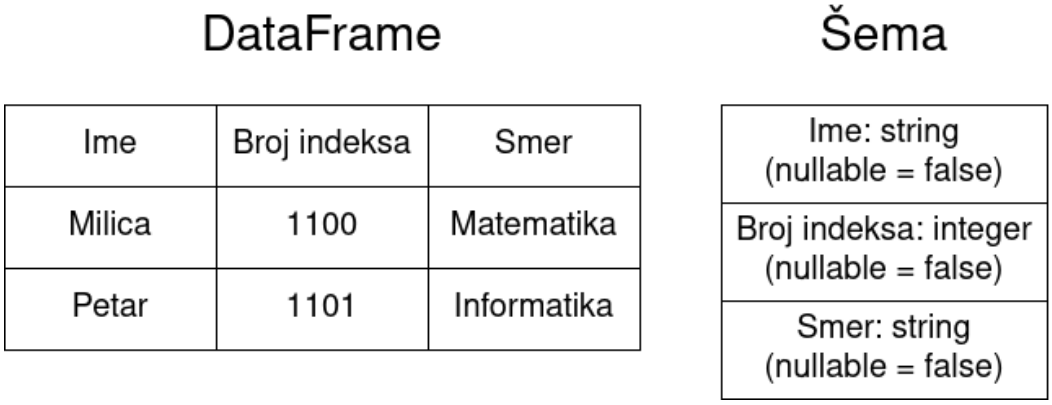
Постоје различити нивои кеширања у зависности од тога у којој врсти меморије се чувају партиције. Уколико је партиције потребно сачувати у меморији, позива се *cache()* или *persist()* са *MEMORY_ONLY* аргументом. Уколико је меморија попуњена, *RDD* се може чувати и на диску. То се постиже прослеђивањем вредности *DISK_ONLY* функцији *persist()*. Овај пристип се не саветује због тога што је често брже рекреирати цео ланац трансформација из почетка, него учитати *RDD* са диска. Функцији *persist()* се могу проследити још три вредности. *MEMORY_ONLY_2* и *DISC_ONLY_2* кеширају *RDD* редом у меморији и диску, али поред тога извршавају репликацију тог *RDD*-ја на још једну машину у кластеру. Вредност *MEMORY_AND_DISC* чува *RDD* у меморији, уколико постоји довољно простора за то. У супротном, кеширање се извршава на диску. [11]

Апстракција *Spark DataFrame*

DataFrame је дистрибуирана колекција налик табели, са дефинисаним редовима и колонама. Свака колона мора имати исти број редова и сваки ред мора имати исти број колона. Поред тога, свакој колони је додељен један тип и тог типа морају бити све вредности које се налазе у њој. [1]

Сваки *Spark DataFrame* садржи метаподатке који описују имена колона и њихове типове. Ти метаподаци се називају шема (енг. *schema*). Шема се може дефинисати експлицитно али се може и аутоматски закључити из података који се налазе унутар *DataFrame*-а. Поред типова, у шеми се налазе информације о томе да ли колона може поседовати *null* вредности. На слици

3.18 је приказан једноставан пример *DataFrame*-а и његове шеме. [1]



Слика 3.18: *Spark DataFrame* и његова шема

Spark нуди велики број типова, који се мапирају у типове програмских језика у којима се он користи. Постоје једноставни типови попут целобројних и децималних бројева и ниски али постоје и сложени, попут низова, мапа и датума. [1]

Трансформације и акције *DataFrame*-а

Све особине трансформација и акција за *RDD* важе и за трансформације и акције *DataFrame*-а. Дакле, трансформације су лење, без бочних ефеката и као резултат се добија иста структура података над којом је трансформација примењена. Ланац трансформација се извршава тек када се над њим позове акција.

Једина разлика је та што *RDD* и *DataFrame* другачије представљају податке, па су им трансформације и акције другачије. Како је *DataFrame* сличан табели у релационим базама, поседује неколико трансформација које су идентичне као наредбе у програмском језику *SQL*. Постоји велики број трансформација које се могу применити над *DataFrame*-ом, међу којима су вероватно најкоришћеније: [1]

- *select(columnNames)*, резултат је нови *DataFrame* са наведеним подскупом колона почетног
- *filter(condition)*, резултат је нови *DataFrame* са редовима почетног који задовољавају задати услов

- *join(otherDataFrame, joinExpression)*, спаја два *DataFrame*-а у један, на основу вредности њихових колона
- *withColumn(newColumnName, expression)*, резултат је нови *DataFrame* са додатом колоном чије се вредности одређују на основу прослеђеног израза

Неке од акција које се могу применити на *DataFrame* су, на пример, *collect()*, која трансформише *DataFrame* у струкуру података програмског језика у коме се *Spark* користи. Акција *show()* се користи за испис *DataFrame*-а на стандардни излаз. Поред њих постоје многеи друге. [1]

Разлика између *DataFrame*-а и *RDD*-ја

Поред различитог начина представљања података, *RDD* и *DataFrame* имају знатне разлике у перформансама. *RDD* се користи за програмирање ниског нивоа, пошто омогућава директан рад са партицијама. Међутим, приликом писања *RDD* трансформација, програмер мора бити веома пажљив када и коју трансформацију примењује, због тога што редослед може значајно да утиче на перформансе. [1]

Са друге стране, редослед примене трансформација код *DataFrame*-а не утиче на брзину извршавања. Разлог томе је што сваки *DataFrame* кôд пролази кроз процес оптимизације, па ће резултат аутоматски бити најбржи могући. За оптимизацију је задужен процес који се зове *Catalyst*. Због перформанси, али и због једноставнијег интерфејса, *DataFrame* је скоро потпуно потиснуо *RDD* из уоптребе. [1]

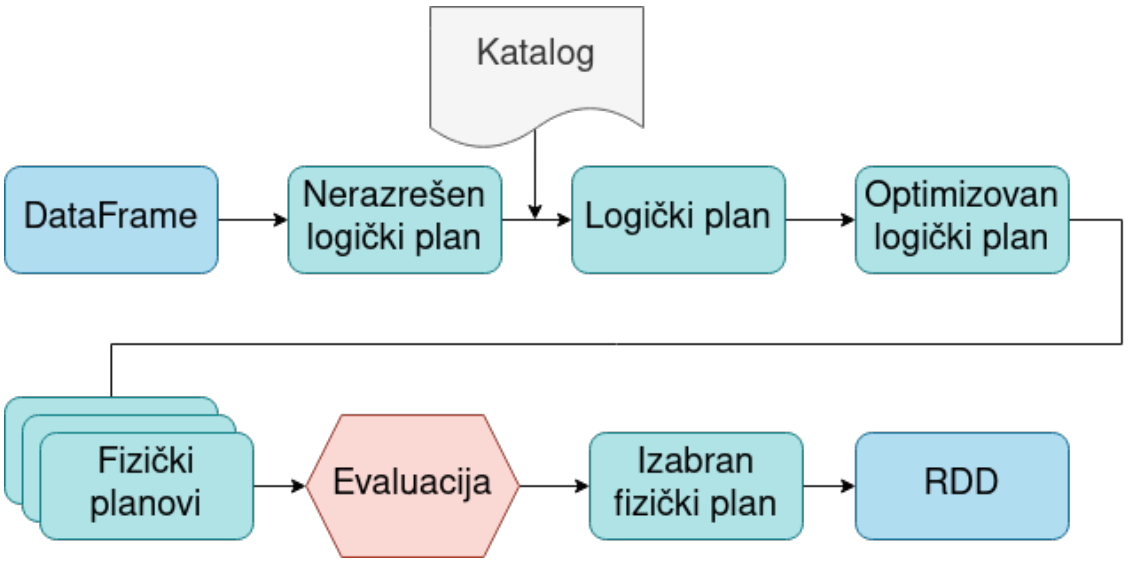
Планови извршавања

Процес извршавања *DataFrame*-а укључује следеће кораке:

1. Кôд *DataFrame*-а је написан
2. уколико је исправан, од њега се прави логички план
3. Од логичког плана се конструише физички план, уз примену оптимизација
4. Добијени физички план се пребацује у *RDD* и извршава се

Ако је *DataFrame* кôд синтаксно исправан, од њега се конструише неразрешен логички план (енг. *unresolved logical plan*) који представља трансформације које треба извршити, али не садржи никакве информације о томе над којим табелама и колононама. Те информације се добијају уз помоћ каталога (енг. *catalog*) у коме се налазе информације о *DataFrame*-овима. Резултат примене каталога на неразрешен логички план је логички план (енг. *logical plan*). Након тога се он оптимизује од стране *Catalyst*-а, који оптимизује логички план тако што га анализира и примењује одређена правила оптимизације на њега. Резултат *Catalyst*-а је оптимизовани логички план (енг. *optimized logical plan*). [1]

Након што се оптимизовани логички план успешно креира, *Spark* у односу на њега конструише неколико физичких планова (енг. *physical plan*). Физички план дефинише на који начин и уз помоћ којих наредби ће се логички план извршити на кластеру. Сви физички планови се након тога евалуирају и бира се онај са најбољим перформансама. Тај физички план се преводи у *RDD* трансформације и извршава на кластеру. Цео процес је приказан на слици 3.19. [1]



Слика 3.19: Ток извршавања *Spark DataFrame*-а

Остале компоненте Спарка

У *Spark*-у, поред *RDD* и *DataFrame* апстракције, постоје још две, *DataSet* и *Spark SQL*. Веома су сличне *DataFrame*-у и пролазе кроз исти процес извршавања, али ипак постоје мање разлике између њих. [1]

Spark SQL се користи за извршавање *SQL* упита над подацима у кластеру. Једина разлика у односу на *DataFrame* је у томе што се синтаксне грешке *SQL* кода појављују тек када се кôд извршава, а не у компилацији, што је случај код *DataFrame*-а. [1]

Разлика између *DataSet*-а и *DataFrame*-а је у провери типова колона. Они се код *DataFrame*-а проверавају са онима у шеми у току извршавања програма, док се код *DataSet*-а то дешава за време компилације. Такође, разлика је у томе што је *DataSet* доступан само у *JVM* базираним језицима, Скали и Јави, док у другима не постоји. [1]

Уз помоћ *Spark*-а се могу конструисати модели машинског учења, преко *Spark MLlib* библиотеке. Она се може користити за препроцесирање, тренирање модела и прављење предвиђања. *Spark* поседује и библиотеке за рад са графовима, *GraphX*, али се за обраду графова на кластеру често користе друга решења. [1]

У *Spark*-у постоје и операције над токовима података. Могуће је закачити се за ток података и над њим примењивати исте трансформације као код *DataFrame*-а. *Spark streaming* омогућава преплату на токове података који настају из веб сокета (енг. *web socket*), локације у фајл систему или од стране *Apache Kafka*-е. [12]

Глава 4

OSM

4.1 Шта је OSM?

OpenStreetMap, скраћено *OSM*, је бесплатна и изменљива мапа света која дозвољава приступ самим мапама, као и подацима које оне садрже. Идеја иза овог пројекта је настанак мапа које се развијају и одржавају од стране заједнице корисника и које ће представљати бесплатну алтернативу неким већ постојећим мапама, попут оних које развија Гугл. [6]

Слика 4.1: Лого *OSM*-а

OSM је иницонално основан 2004. године од стране Стива Коуста (енг. *Steve Coast*) са идејом да мапира Уједињено Краљевство. У следећим годинама је пројекат постао глобалан и сада садржи податке целог света. [6]

4.2 Елементи

Елементи су основне јединице *OSM* моделовања података физичког света. Постоје три врсте и то су:

- чворови (енг. *nodes*)
- путање (енг. *ways*)
- релације (енг. *relations*)

Сваки од елемената може имати придружен један или више тагова (енг. *tag*) чија је улога бољи опис елемента коме припада. Елементи *OSM* скупа се могу представити помоћу *XML* записа од којих сваки има засебан *XML* таг унутар кога се налазе атрибути. [6]

Чворови

Чвор представља локацију на Земљиној површини и састоји се од две координате, географске дужине и географске ширине као и идентификатора који је јединствен. Један чвор се може користити да дефинише неки објекат на мапи, попут, на пример, клупе или статуе. [6]

У *XML* језику чворови су представљени *XML* тагом *node* унутар кога су угњеждени *OSM* тагови чвора.

```
<node id="25496583" lat="51.5173639" lon="-0.140043"
   version="1" changeset="203496" user="80n" uid="1238"
   visible="true" timestamp="2007-01-28T11:40:26Z">
    <tag k="highway" v="traffic_signals"/>
</node>
```

Путање

Путање су уређене листе које садрже између 2 и 20000 чворова и представљају линеарне објекте на мапи, попут путева или река. Такође, могу представљати и разне врсте површина, попут шума. У том случају су представљени листом којој су први и последњи елемент исти чвор. [6]

Путање се *XML* форматом представљају као листа идентификатора чворова које путања садржи, као и њених тагова.

```
<way id="5090250" visible="true" timestamp="2009-01-19
   T19:07:25Z" version="8" changeset="816806" user="Blumpsy
   " uid="64226">
```

```
    <nd ref="822403"/>
    <nd ref="21533912"/>
    <nd ref="821601"/>
    <nd ref="21533910"/>
    <nd ref="135791608"/>
    <nd ref="333725784"/>
    <nd ref="333725781"/>
    <nd ref="333725774"/>
    <nd ref="333725776"/>
    <nd ref="823771"/>
    <tag k="highway" v="residential"/>
    <tag k="name" v="Clipstone Street"/>
    <tag k="oneway" v="yes"/>
  </way>
```

Релације

Релације су структуре које представљају некакав однос између елемената, чворова, путања или других релација. Значења релација могу бити разна па су због тога оне описане таговима. Обично, свака релација поседује таг који се зове *type* и сваки други таг релације се интерпретира на основу вредности тог тага. [6]

Приказ релација у *XML* формату се врши приказом чланова те релације и њених тагова.

```
<relation id="56688" user="kmvar" uid="56190" visible="
   true" version="28" changeset="6947637" timestamp="
   2011-01-12T14:23:49Z">
  <member type="node" ref="294942404" role=""/>
  ...
  <member type="node" ref="364933006" role=""/>
  <member type="way" ref="4579143" role=""/>
  ...
  <member type="node" ref="249673494" role=""/>
  <tag k="name" v="Linie 123"/>
  <tag k="network" v="VVV"/>
```

```
  <tag k="operator" v="Regionalverkehr"/>
  <tag k="ref" v="123"/>
  <tag k="route" v="bus"/>
  <tag k="type" v="route"/>
</relation>
```

Тагови

Као што је већ речено, тагови представљају опис неког елемента. Сваки елемент може имати нула, један или више тагова. Чине га две вредности, кључ, који мора бити јединствен унутар елемента ког таг описује, и вредност. [6]

Заједнички атрибути елемената

Као што се из приказаних *XML* записа може закључити, сваки елемент има некакве атрибуте који га описују. Заправо, постоје одређени атрибути се налазе у сваком елементу, а то су:

- *id*, јединствен идентификатор елемента
- *user*, име корисника који је изменио елемент
- *uid*, идентификатор корисника који је изменио елемент
- *timestamp*, време последње промене елемента
- *visible*, знак који показује да ли је елемент избрисан
- *version*, тренутна верзија елемента. Почетна вредност је 1 и сваки пут када се изврши некаква модификација тај број се инкрементира
- *changeset*, идентификатор скупа промена у коме је елемент измењен

Глава 5

App

5.1 Opis

ovde opis appa i sta se trazi od nas

5.2 Cloud

шта је cloud и које услуге нуди. EMR, EC2, S3 нпр мало детаљније. За ово или ова секција или одвојена, након дистрибуираних система.

5.3 Arhitektura aplikacije

opis komponenti

5.4 Подаци

skupovi podataka koji se koriste mada je vecina toga vec napisana ranije.

5.5 Obrada OSM skupa Sparkom

koje spark transformacije su primenjene na koje podatke

5.6 PolyContains

algoritmi korisceni za poly contains i rezultati (graficki prikaz)

5.7 Rezultat

dobijeni rezultati aplikacije, prikaz nekih selekcija itd

Глава 6

Закључак

zakljucak rada

Библиографија

[1] Bill Chambers and Matei Zaharia. *Spark: The definitive guide*. 2018.

[2] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. 2004. online at: https://research.google/pubs/pub62/.

[3] Apache foundation. Apache hive. online at: https://hive.apache.org/.

[4] Apache foundation. Apache pig. online at: https://pig.apache.org/.

[5] Apache Foundation. Hdfs architecture guide. online at: https://hadoop.apache.org/docs/.

[6] OpenStreetMap Foundation. Openstreetmap wiki. online at: https://wiki.openstreetmap.org/.

[7] Arne Horst. Amount of data created, consumed, and stored 2010-2025. 2021. online at: https://www.statista.com/statistics/871513/worldwide-data-created/.

[8] Lex Spoon Martin Odersky and Bill Venners. *Programming in Scala, First edition*. 2008.

[9] Marcus Krotzsch Pascal Hitzler and Sebastian Rudolph. *Foundations of Semantic web technologies*. 2010.

[10] Howard Gobioff Sanjay Ghemawat and Shun-Tak Leung. The google file system. 2003. online at: https://research.google/pubs/pub51/.

[11] Apache Spark. Rdd programming guide. online at: https://spark.apache.org/docs/latest/rdd-programming-guide.html.

[12] Apache Spark. Structured streaming programming guide. online at: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html.

[13] Garry Turkington. *Hadoop beginner's guide*. 2013.

[14] Garry Turkington and Gabriele Modena. *Learning Hadoop 2*. 2015.

Биографија аутора

Вук Стефановић Караџић (*Тршић, 26. октобар/6. новембар 1787. — Беч, 7. фебруар 1864.*) био је српски филолог, реформатор српског језика, сакупљач народних умотворина и писац првог речника српског језика. Вук је најзначајнија личност српске књижевности прве половине XIX века. Стекао је и неколико почасних доктората. Учествовао је у Првом српском устанку као писар и чиновник у Неготинској крајини, а након слома устанка преселио се у Беч, 1813. године. Ту је упознао Јернеја Копитара, цензора словенских књига, на чији је подстицај кренуо у прикупљање српских народних песама, реформу ћирилице и борбу за увођење народног језика у српску књижевност. Вуковим реформама у српски језик је уведен фонетски правопис, а српски језик је потиснуо славеносрпски језик који је у то време био језик образованих људи. Тако се као најважније године Вукове реформе истичу 1818., 1836., 1839., 1847. и 1852.