УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ

Давид Гавриловић

ОБРАДА ГЕОПРОСТОРНИХ ПОДАТАКА КОРИШЋЕЊЕМ ДИСТРИБУИРАНИХ СИСТЕМА

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: Обрада геопросторних података коришћењем дистрибуираних система

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 нежну 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 Феђа чупа за косу Миљу, она јури Живу, али њега хоће и Даца. Док је Фехим у џипу журно љубио Загу Чађевић, 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, геџо, 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 дворишта. Зашто гђа Хаџић лечи живце: њена љубав пред фијаском? Јеж хоће пецкањем да вређа љубичастог џина из флаше. Џеј, љубичаст зец, лаже: гађаће одмах поквашен фењер. Плашљив зец хоће јефтину дињу: 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 *Scala*

Скала (енг. Scala) је виши програмски заснован на функционалној и објектној 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. [5]

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*. [5]

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. Дакле, 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. [5]

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. [5]

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. Наравно, поред тога, Скала омогућава писање функционалног кода без бочних ефеката и са имутабилним структурама података, па се може користити и на тај начин. [5]

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. [5]

Међутим, иако се компајлирају на исти начин, приграми написани у језику *Scala* 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 је и до мање вероватноћа прављења грешкака. [5]

Један од многих примера како *Scala* 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. [5]

Статичка типизираност

Статичка типизираност значи да се типови промењљивих 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 неку 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. [5]

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 следеће две линије кода (2.3) имају еквивалентно понашање.

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

// primer 2
val x = 10
```

Кôд 2.3: Декларација типова

Наравно, исто важи и за било који други *Scala* тип, не само за целобројни.

Scala 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. [5]

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 променљива названа **res0** типа *Int* у којој ће се налазити резултат унетог израчунавања. Новонастала променљива се може користити у наставку извршавања. [5]

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

Кôд 2.6: Коришћење резултатских променљивих

Уколико је потребно само исписати вредност у конзолу без креирања нове променљиве 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*. [5]

Поред њих постоје и уграђени сложени типови попут ниске (*String*), н-торке (*Tuple*) или низа (*Array*) и других. Како је Скала објектни 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 које на тај тип можемо 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 дефинишемо кључним речима *var* и *val*. Разликују се по томе што се вредност промењивих дефинисаних са *val* не могу мењати, док је код оних дефинисаним са *var* то могуће, све док је нова додељена вредност компатибилног типа. [5]

У наставку (примери 2.8, 2.9 и 2.10) су приказани примери дефиниција 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 res4: 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. [5]

```
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 функције 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, може се изоставити које означавају тело 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 рекурзију. Такође, функција не мора да враћа никакву вредност. У том случају је повратни тип означен са *Unit*. [5]

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 Објектна оријентисана парадигма

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

Класе

Као и у Јави, у Скали класа представља нацрт ког се инстанцирају 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*), које се једним именом називају чланови (енг. *members*). Поља су промењљиве 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. [5]

```
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.

```
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).

```
// 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) // instanciranje nadklase
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.

```
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

У Скали не постоје статична поља. Уместо тога постоје синглтон објекти (енг. *singleton object*). Објекти се дефинишу на исти начин као и класе с тим што се користи кључна реч *object* уместо *class*. Сви чланови објекта се понашају као статични чланови у Јави. [5]

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

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

Уколико објекат дели своје име са неком класом, а при томе се налазе у истом фајлу, тај објекат називамо објекат сапутник (енг. *companion object*). Тада класа и објекат могу да приступају приватним пољима оног другог.

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

```
scala> object Main {
     |  def main(args: Array[String]): Unit = {
     |      print("Hello :3")
     |  }
     | }
```

Traits

Trait је основна јединица наслеђивања у Скали. Унутар њега наводимо поља и методе које касније можемо користити у класама које га имплементирају. Разликују се од наслеђивања по томе што класа може наследити само једну класу док са друге стране може наследити више од једног *trait*-а. [5]

Дефинише се на исти начин као и класа с тим што се уместо кључне речи *class* користи *trait*.

```
scala> trait MyTrait
trait MyTrait
```

Додаје се 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
```

Уколико класа којој додељујемо *trait* већ наслеђује неку класу, то се мора урадити преко кључне речи *with*. Сваки нови *trait* који се додаје у овом случају се мора додати након нове речи *with*. [5]

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

Поред овога, *trait* се може користити и као тип, с тим што вредност променљиве која је тог типа мора бити нека 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
```

Све ово значи да је Скала *trait* веома сличан Јавином интерфејсу, с разликом у томе да *trait* може садржати и дефиниције метода, а не само декларa-

ције. Међутим, *trait* је више од тога. Штавише, унутар њега можемо урадити све исто што можемо и у Скала класи.

Case класе

Поред обичних, у Скали постоји још једна посебна врста класе названа *case* класа. Постоје три разлике између ове врсте класа и обичне:

- 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)
```

Највећа предност ових класа је та да се могу користити у једном конструкту специфичном за функционалне језике, *pattern matching*-у, о коме ће речи бити касније. [5]

2.8 Скала колекције

Низови

Скала нуди велики број уграђених колекција, мутабилних и имутабилних. Скала низ (енг. *Array*) је мутабилна структура која представља низ података. Мутабилна је због тога што, иако се број елемената у низу не може мењати, могу се мењати саме елементи. Сваки низ садржи елементе само једног типа и може се креирати навођењем иницијалних елемената или његове дужине. [5]

```
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)
```

Приметимо да у другом случају инстанцирамо нову класу уз помоћ наредбе *new* док у првом то не радимо. Разлог је то што се у првом случају позива метод *apply()* који креира низ за нас. Приметимо и то да је у првом случају Скала закључила тип низа, на основу његових елемената, док смо у другом то морали експлицитно да назначимо. [5]

Елементу низа приступамо као у Јави, само што уместо [] користимо (). На сличан начин и мењамо један елемент низа.

```
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)
```

Листе

Уколико желимо да користимо имутабилалну колекцију елемената истог типа можемо користити Скала листе. Разлика листе у Скали у односу на ону у Јави је та што је Скала листа увек имутабилна, док Јава листа може бити мутабилна. Листу можемо инстанцирати навођењем њених елемената. Приступ елементу листе се врши исто као у низу, али мењање елемената није дозвољено. [5]

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

scala> l(0)
```

```
val res59: String = a

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

Листе се спајају оператором :::. Напоменимо да се коришћењем овог оператора не додају елементи једне листе на другу, већ се креира тотално нова листа. Овај метод можемо користити да надовежемо и више од једне листе. [5]

```
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)
```

Још један веома користан оператор је ::, који као аргументе прима листу и елемент од којих прави 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)
```

Још један начин иницијализације листе је надовезивање елемената на празну листу оператором ::. Напоменимо да се у Скали празна листа означава кључном речи *Nil*.

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

Н-торке

Имутаблина колекција која садржи елементе различитог типа се назива н-торка, *Tuple*. Ова структура података је корисна када, на пример, желимо да вратимо више различитих вредности из неке функције. *Tuple* се инстанцира веома лако, само треба навести његове елементе између заграда. Елементу приступамо оператором _X где је X редни број елемента унутар н-торке. [5]

```
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)
```

Скупови и мапе

Поред поменутих, у Скали постоје и друге структуре података. Скуп користимо када желимо колекцију која садржи све елементе истог типа, с тим што сваки елемент колекције мора бити јединствен. Постоје две врсте скупа, имутабилни, *collection.immutable.Set* и мутабилни, *collection.mutable.Set*.

Инстанцирају се на исти начин, навођењем елемената. Уколико током навођења наведемо неки елемент више од један пут нећемо добити грешку већ ће се дупликат уклонити. Да бисмо додали елемент у скуп користимо оператор +. Код мутабилних скупова + додаје вредност на постојећи скуп, док код имутабилних креира тотално нови скуп. [5]

Поред скупова, Скала има структуре за рад са кључ-вредност паровима. Такве структуре података су назване мапе и у Скали постоје мутабилне, *collection.mutable.Map* и имутабилне, *collection.immutable.Map*, које су подразумевани тип.

Мапу дефинишемо низом кључ $->$ вредност парова који могу бити било ког типа. Једино битно је да су сви кључеви и све вредности међусобно 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 res36: 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 res40: Array[Int] = Array(1, 2)
```

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

```
scala> // mutable

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 res52: scala.collection.mutable.Map[String,String] =
   HashMap(k1 -> vred1, k2 -> vred2)

scala> // 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]
```

2.9 Pattern matching

Скалин *pattern matching* се може посматрати као побољшана верзија *switch* наредбе у Јави. Састоји се од селектора, који представља некакав израз или променљиву, кључне речи *match* и низа случајева унутар који могу да одговарају селектору. Сваки израз се састоји од знака => који се налази између неке вредности и кључне речи *case* са леве стране и израза који ће бити резултат са десне. [5]

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

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

- *match* наредба увек резултује неком вредношћу
- Када пронађемо одговарајућу вредност у Скали, ту се заустављамо и не разматрамо даље
- Уколико ниједна вредност не одговара селектору, појавиће се *MatchError*. То значи да увек морамо да пазимо на то да покријемо све могуће вредности селектора

Џокери

У Скали постоји џокер (енг. *wildcard*) који одговара било којој вредности селектора. Његова ознака је _. Може се користити у случају да се селектор спаја са неком вредношћу или, уколико је селектор класа, уколико се врши спајање поља те класе са неком вредношћу. Погледајмо примере: [5]

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

scala> x match {
     | case 10 => print("x je 10")
     | case _  => print("x nije 10")
     | }
x je 10

scala> case class MyClass(val x: Int)
class MyClass

scala> val mc = MyClass(10)
val mc: MyClass = MyClass(10)

scala> mc match {
     | case MyClass(10) => print("x unutar klase je 10")
     | case MyClass(_)  => print("x unutar klase nije 10")
     | }
x unutar klase je 10
```

Примери коришћења

Скалин *pattern matching* је веома моћан механизам и може се користити у великом броју случајева. Можемо вршити 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"
     | }
```

Такође се користи и за поклапање променљивих. У следећем случају, променљива *something* ће одговарати било којој вредности, која ће се уз помоћ ње пренети на десну страну знака =>. [5]

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

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

Користи се и за поклапање типова селектора, уколико желимо да дефинишемо да за различите типове користимо различито понашање.

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

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

Поред свих наведених примера, *pattern matching* се користи у још великом броју случајева. На пример, користи се за поклапање секвенци, н-торки и класа. Погледајмо пример где функција *generalSize(x: Any)* има различито понашање у односу на то ког типа је њен аргумент. [5]

```
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 res8: Int = 11

scala> generalSize(12)
val res9: Int = -1
```

Глава 3

Дистрибуирани системи

3.1 Увод

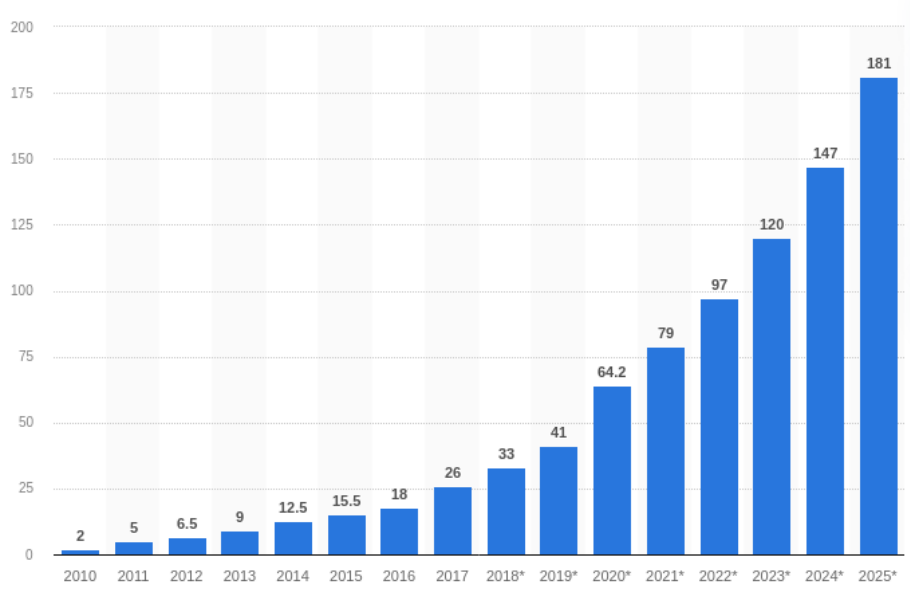
У данашње време се генерише огромна количина података. Друштвене мреже, видео садржај и куповина преко интернета су само неке од ствари које на дневном нивоу производе петабајте података, а у будућности се очекује пораст тог тренда.

Према истраживању приказаном на једном од водећих интернет платформи за податке који се користе у пословању, количина података који производимо се тренутно може мерити у десетинама зетабајта (енг. *zetabyte*) [1]. Исто истраживање приказује да ће се наредних година тај број удвостручити. [4]

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 испоставило да такав приступ има своје лимите.

[1] милијарда терабајта



Слика 3.1: количина података по години у зетабајтима

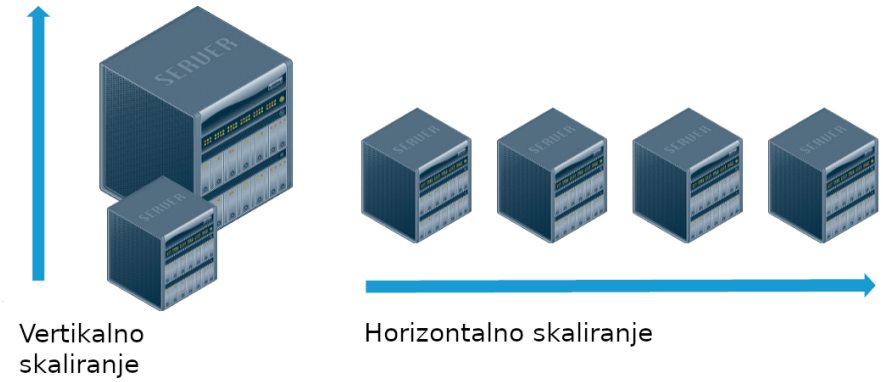
3.2 Скалирање система

Постоје два начина скалирања уређаја који врше обраду података. Први начин је вертикално скалирање (енг. *vertical scaling* или *scale-up*). У овом приступу ми унапређујемо једну машину, на пример додајемо јој више меморије или појачавамо снагу процесора. Предности овог приступа је то што након што унапредимо машину не морамо да мењамо логику апликација које се на њој извршавају. Али негативне ствари су те што постоји некакво ограничење до ког можемо унапредити машину (ипак морамо обрадити егзабајте података), као и то да морамо да обратимо посебну пажњу на то шта се дешава ако систем доживи некакву грешку и неочекивано престане са радом. [11]

Други приступ је хоризонтално скалирање (енг. *horizontal scaling* or *scale-out*). У овом случају не унапређујемо једну машнину, већ, уколико нам је потребна додатна снага, додајемо нову машину у систем. Добра ствар је та што је често јефтиније додати неколико нових машина у систем него унапредити процесор неколико пута на истој машини. Још једна веома добра ствар је ефикасност. Када имамо неколико машина можемо 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. [11]

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



Слика 3.2: Врсте скалирања

3.3 Особине дистрибуираних система

Идеја је да свака машина у систему обрађује један део података и да на тај начин допринесе коначном резултату. То значи да машине морају да комуницирају једна са другом и да ће неки подаци можда затребати свакој машини у систему. То може довести до тога да се уређаји такмиче међу собом за приступ 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 доживи некакав проблем и да због тога престане да функционише, што би изазвало престанак рада целог система. [11]

Због тога, систем би требало да функционише тако да уређаји који су у њему раде независно од других уређаја истог система, као и да престанак рада једне машине не утиче на систем у целини. Другим речима, требало би направити такав систем који очекује да се фаталне грешке дешавају. У овим системима је акценат на софтверу, а не на хардверу. Наиме, идеја је да се систем може направити од уређаја који су масовно доступни. Такође, идеја је

да се избегава трансфер података међу уређајима. Дакле, подаци би, уколико је то могуће, требало да се обрађују на машини на којој се налазе. [11]

3.4 Хадуп

Први успешан систем који поседује претходно наведене карактеристике је остварила компанија Гугл (енг. *Google*) која је 2003. године објавила научни рад на ту тему. У том научном раду је представљен дистрибуирани фајл систем, назван Гугл фајл систем (енг. *Google file system*) или скраћено ГФС (енг. *GFS*). Идеја иза овог пројекта написаног у програмском језику *C++* је да се користи за складиштење великих количина података. [8]

Већ следеће године, Гугл је објавио нови научни рад о парадигми за ефикасну обраду велике количине података на кластеру. Парадигма је названа Мап редјус (eng. *Map Reduce)* и њена намена је да се користи за обраду података складиштених у ГФС-у. [2]

Недуго након тога, уз помоћ научних радова компаније Гугл, настао је пројекат отвореног кода (енг. *Open source*) назван Хадуп (енг. *Hadoop*) са идејом да имплементира карактеристике које имају гуглови ГФС и Мап Редјус и да се као такав користи за складиштење и ефикасну обраду падатака на кластеру сачињеном од релативно јефтиних машина. Највећи делови Хадупа су ХДФС (енг. *HDFS*) и Мап редјус (енг. *Map Reduce*), скраћено МР (енг. *MR*) који су заправо јавно доступни еквиваленти Гуглових технологија. [11]

Укратко, ХДФС 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. [11]

МР је парадигма за обраду података, која се састоји из два дела названа Мап (енг. *map*) и Редјус (енг. *reduce*), по којима је парадигма и добила име. У оба дела се над подацима који се налазе у ХДФС-у врши некаква обрада података. Напоменимо да се и сама обрада извршава на ХДФС-у, на истом месту где се подаци и налазе, дакле избегава се беспотребно премештање на неку другу машину. [11]

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: ХДФС и МР

Поред поменуте две компоненте, постоји и трећа, а то су ХДФС апликације. Оне се надовезују на ХДФС и МР тако што их користе за редом складиштење и обраду. Најпознатије од њих су Хајв (енг. *Hive*) и Пиг (енг. *Pig*), али поред њих наравно постоје и многе друге, само мање заступљене. На слици 3.4 можемо видети компоненте Хадупа. [12]

Слика 3.4: Упрошћен приказ Хадупа

3.5 ХДФС

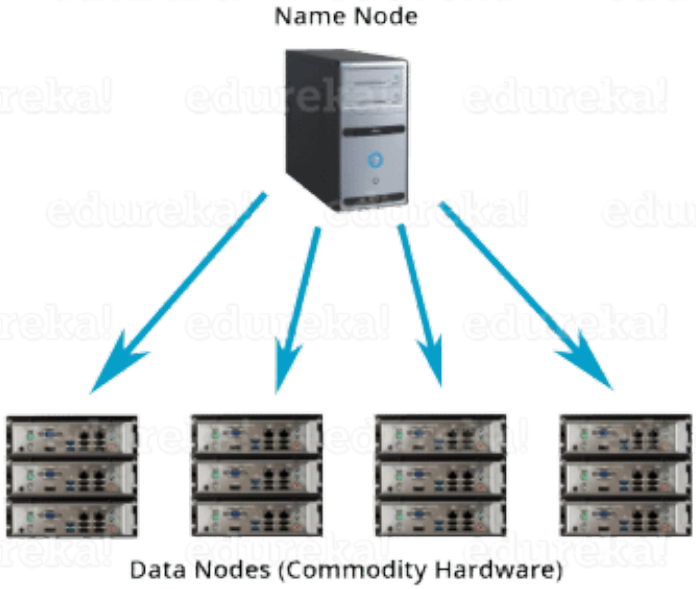
ХДФС, скраћено од Хадупов дистрибуирани фајл систем (енг- *Hadoop distributed file system*) је, као што му је у имену и написано, дистрибуирани систем, што значи да складишти податке на више машина које ћемо у даљем тексту звати чворови (енг. *nodes*).

Врсте уређаја ХДФС-а

Постоје две врсте чворова, именски (енг. *name node*) и дата (енг. *data node*) (слика 3.5). Сваки ХДФС систем има један 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 промене фајла као и правила приступа. [3]

Друга врста је дата чвор и њих се у систему обично налази више од једног. На њима се заправо складиште фајлови у систему али се поред тога користе и за компутације. Такође, дата чворови су задужени за операције над фајловима као што су читање, мењање и брисање. Они ће урадити неку од тих операција само када им именски чвор то нареди. [3]

Уколико имамо неког клијента који жели да приступи ХДФС-у, он ће прво комуницирати са именским чвором и од њега затражити фајлове који му требају. Након тога ће именски чвор проверити да ли клијент има потребне пермисије и ако их има, послаће му локацију фајла у фајл систему. Тек након тога се може извршити жељени приступ.



Слика 3.5: Врсте чворова у ХДФС-у

Особине

Сваки фајл у ХДФС-у је подељен на делове које зовемо блокови, чија је величина обично 128 мегабајта. Блокови се често не налазе на истим чворовима у систему, што значи да један фајл чувамо 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. Ако се то деси, сви блокови на тој машини ће нестати. ХДФС очекује ово и због тога се сваки блок реплицира неколико пута и онда се оригинални блок и његове реплике распореде по систему. Ако један од блокова фајла неочекивано нестане, увек можемо приступити једној од његових реплика. Напоменимо још једном да ће се реплике чувати на дата чворовима, док ће се информације о томе ком фајлу реплике припадају налазити на именским чвору. [3]

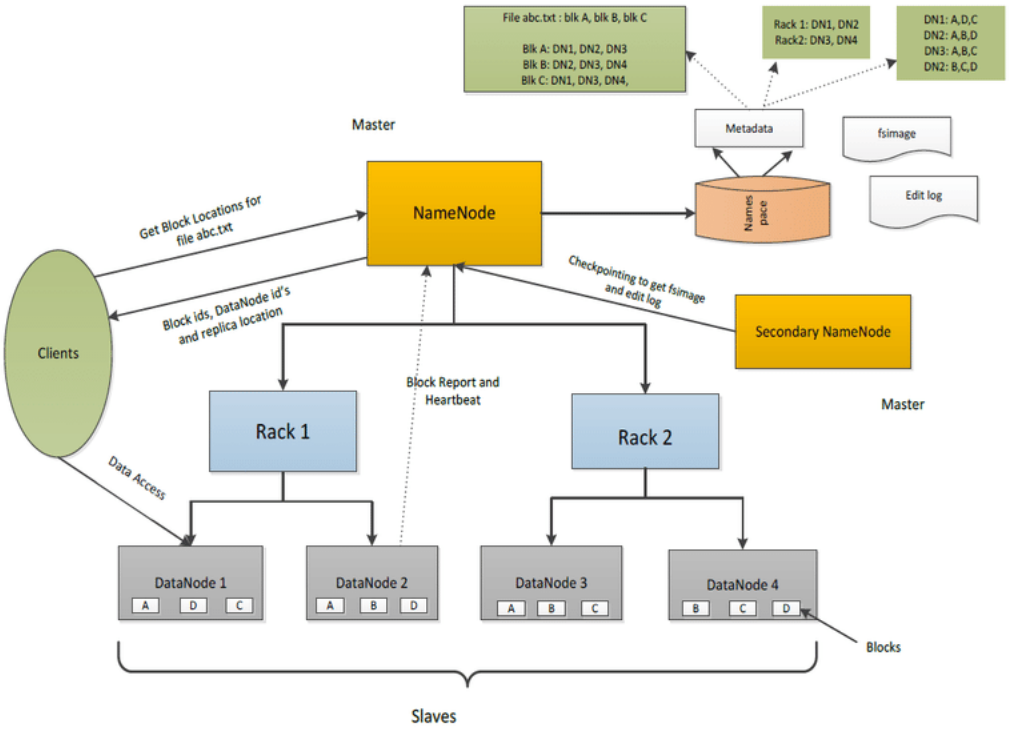
Блок ће се увек реплицирати одређен, фиксан, број пута. Дата чворови повремено шаљу сигнале именском чвору о доступности реплика. На тај начин ће именски чвор увек знати колико је пута сваки блок реплициран у систему и на основу тога може да, уколико тај број падне испод неког задовољавајуће вредности, направи нове реплике тог блока. [3]

Поменули смо да ХДФС очекује пад дата чворова и губитак података на њима. Али шта се дешава ако се проблем деси на најбитнијем чвору у систему, именском, и он неочекивано престане са радом? Такви проблеми се заобилазе тако што се чувају резервне копије именског чвора због којих се и у случају престанка његовог рада не губе никакве информације. Напоменимо да се резервне копије праве у одређеним временским интервалима да би подаци на њима били ажурни. Поменути концепти су приказани на слици 3.6.

Претходно поменуте особине су битне за целокупну робусност система, односно поузданости података чак и у случајевима када се десе некакви проблеми.

ХДФС је систем за кога важи пиши једном, читај више пута (енг. *write-once, read-many (WORM)*). Дакле, када поставимо фајл на ХДФС не можемо га мењати. Ако желимо да променимо фајл морамо да поставимо тотално нови фајл уместо старог. Иако то није ефикасно, 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 ни потребе или се то барем неће често дешавати. [11]

Такође, још једна од особина ХДФС-а је та да је конструисан да има добре



Слика 3.6: ХДФС компоненте

перформансе у случајевима када нам је потребан велики проток података, на пример у случају читања великих фајлова. [11]

3.6 Мап Редјус

Као што је већ наведено, MP је парадигма која се користи за обраду података који су складиштени у ХДФС-у. Користи подели и завладај (енг. *divide and conquer*) приступ приликом обраде података са идејом да више машина паралелно обрађује један њихов део. На тај начин се, на пример, обрада података која би трајала 1000 минута, ако се паралелизује на 1000 машина, могла свести на обраду која траје само један минут. [11]

Парадигма је заснована на концептима из функционалног програмирања и функцијама које се често користе у обради низова и листи. Те фунције се зову мап и редујс. Прва од постојеће листе креира нову тако што на сваки елемент листе примени неку фунцију и од њега направи нови елемент. Друга од целе листе конструише једну вредност. На истим принципима фунционише и мап редјус парадигма.

МР обрађује податке у неколико фаза. Прво, подаци се читају из ХДФС-а и након тога се прослеђују машинама које називамо мапери (енг. *mappers*). Те машине паралелно производе скуп привремених података који се након тога распоређују, сортирају и шаљу машинама које називамо редјусери (енг. *reducers*). Фаза која распоређује податке се назива фаза мешања и сортирања (енг. *shuffle and sort*). Задатак редјусера је да приме некакав скуп података и да паралелно произведу једну вредност од истих. На самом крају се резултат свих редјусера комбинује и добија се резултат читавог МР процеса, другачије званог и МР задатак (енг. *task*). Напоменимо да је могуће комбиновати МР задатке тако да излаз из једног буде улаз у други. [12]

Мап Редјус из аспекта функција

Из функционалног аспекта, мап и редјус фазе можемо посматрати на следећи начин. Нека имамо листу почетних података у (кључ, вредност) формату. Због једноставности ћемо за поменути формат користити ознаку (к, в). Дакле, током мап фазе се подаци из листе парова читају, деле у мање подлисте на које се паралелно примењује мап функција коју дефинишемо. Паралелизам се постиже тако што се свака подлиста обрађује на засебној машини (маперу). Мап фаза као улаз прима листу (к, в) парова и производи нову листу истог формата. [12]

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

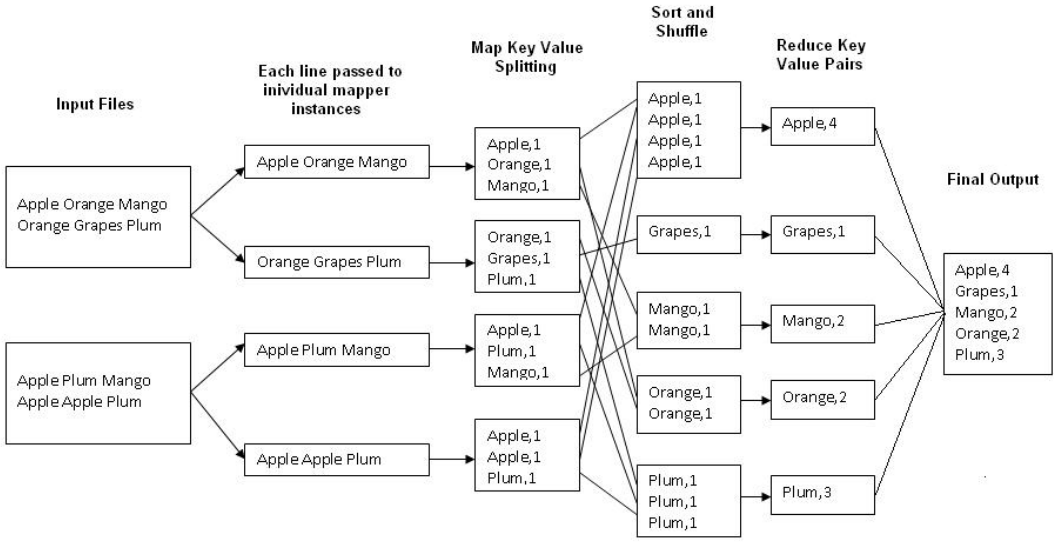
Након тога се те нове листе групишу тако што се за сваки кључ прави једна група која ће бити процесуирана од стране једне машине (редјусер). Дакле, није могуће да се подаци који имају различите кључеве нађу на истој машини у истом тренутку. Тај већ поменути део обраде је *shuffle and sort*.

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

На крају се на сваку од група паралелно примењује редјус функција која производи једну вредност за сваку групу. Дакле редјус прима кључ и листу вредности које му одговарају и као резултат производи једну вредност формата (кључ, вредност). [12]

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

Тако добијен резултат се уписује у ХДФС. Цео процес је приказан на слици 3.7 где је као пример представљена MP апликација која пребројава број појављивања сваке речи у тексту. [12]



Слика 3.7: Пример MP апликације

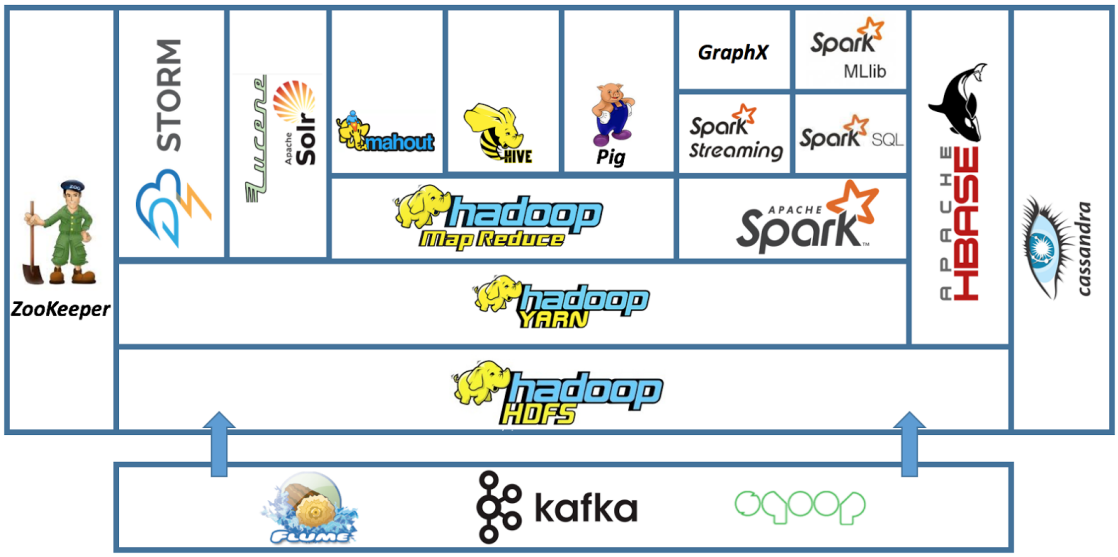
У MP апликацијама задатак програмера је да опише како ће се извршавати мап и редјус фазе, док ће се Хадуп систем побринути за све остало: читање података, сортирање, паралелизацију, координацију и извршавање послова. [11]

Напоменимо и то да је ова парадигма 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. [11]

3.7 Остале компоненте Хадупа

У овој секцији ћемо само напоменути неке од технологија које користе ХДФС и које заједно са њим чине Хадуп екосистем (енг. *Hadoop ecosystem*). Те технологије имају широке области примене. Неке од њих су преговарач

ресурса Јарн (енг. *Yarn*) о коме ћемо у више детаља писати у следећој секцији, Кафка (енг. *Kafka*), апликација која ради над токовима података, Пиг (енг. *Pig*) и Хајв (енг. *Hive*) које су намењене за обраду података и имплементиране су користећи МР. Поред њих постоје и многе друге, као на пример Престо (енг. *Presto*), Хју (енг. *Hue*), Флум (енг. *Flume*) и друге, али оне, као ни операције за које су намењене, нису фокус овог рада. Приказ малог дела Хадуп екосистема је приказан на слици 3.8.



Слика 3.8: Хадуп екосистем

3.8 Апачи Јарн

У првој верзији Хадупа, мап редјус је заправо имао две намене. Прва је била већ поменута обрада велике количине података за коју је МР примарно и намењен, док је друга била алокација и управљање ресурсима који су МР апликацији били потребни као и заказивање МР таскова. Такав систем је конструкцију апликације које користе МР учинио веома тешким. Због тога је одлучено да МР буде намењен само обради података, док је друга улога дата другој апликацији коју би МР користио. [12]

Резултат је менаџер ресурса (енг. *resource manager*) отвореног кода назван Јарн (енг. *Yarn, yet another resource negotiator*) који распоређује таскове апликација које користе Хадуп, али и алоцира ресурсе који су тим апликацијама

потребни. Конструисан је тако да није специфичан само за MP, већ пружа интерфејс ка Хадупу разним апликацијама међу којима је и Спарк, о коме ће пречи бити касније. [12]

Слика 3.9: Апачи Јарн

Архитектура Јарна

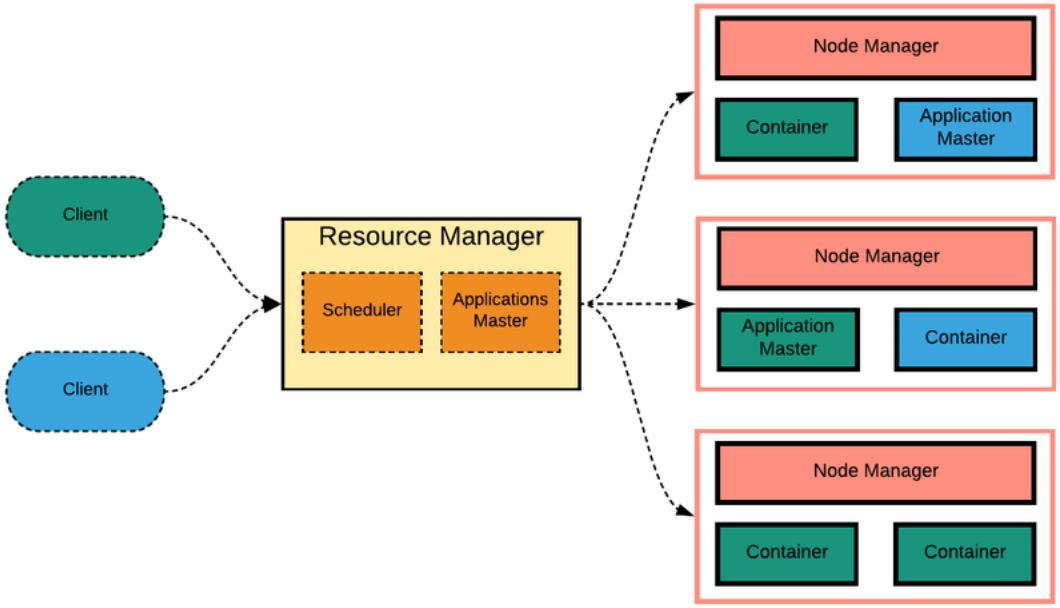
Као што је већ напоменуто, Јарн се не бави обрадом података, већ му је улога да апликацијама које обрађују податке обезбеди ресурсе потребне за њихово извршавање. Направљен је тако да може да подржи разне врсте апликација. [12]

Конструисан је од две главне компоненте, менаџера ресурса (енг. *resource manager*) у даљем тексту РМ и од менаџера чворова (енг. *node manager*) у даљем тексту НМ. Улога првог је да управља ресурсима кластера, док је намена другог да управља ресурсима машине на којој је покренут. То значи да ће кластер имати један РМ и више НМ. Заједно, они ће управљати контејнерима (енг. *container*), апстракцијом меморије, процесорске снаге и И/О операција потребних да би се извршио један део апликације на кластеру. [12]

Улога Јарна је само да обезбеди ресурсе и распореди извршавање таскова на кластеру. Све остало, попут мониторинга, праћења прогреса апликација, обраде грешака и сличног је имплементирано у коду апликације која га користи. Идеја иза тога је да Јарн буде што је више могуће независан да би различите врсте апликација могле да га користе. [12]

Апликација коју покрећемо преко Јарна се састоји из два дела. Први је код који треба извршити док је други назван мастер апликације (енг. *application master*), даље АМ. Његова улога је да управља апликацијом. Као што је у

прошлом пасусу поменуто, Јарн се не бави мониторингом и прогресом апликације која је покренута, већ је то улога саме апликације. То апликација успева да уради преко мастера апликације и Јарн нема информацију о томе на који начин је успостављена комуникације између AM и кода који се извршава. Приказ архитектуре је приказан на слици 3.10. [12]



Слика 3.10: Архитектура Јарна

Апликације се, користећи Јарн, на кластеру покрећу преко клијента. Када започнемо Јарн апликацију, клијент прво комуницира са менаџером ресурса и преко њега захтева иницијални контејнер на коме ће покренути мастер апликације. У већини случајева се мастер апликације покреће унутар једног контејнера у кластеру, исто као што се покреће и код саме апликације. Након тога почиње комуникација између покренутог мастера апликације и менаџера ресурса. Та комуникација се одвија тако што AM захтева од RM контејнере који су потребни апликацији и менаџер ресурса након тога шаље податке AM о контејнерима које апликација може да користи. Користећи те информације, AM комуницира са менаџерима чворова који се налазе на истим машинама као и захтевани контејнери и пружа им спецификације о апликацији која треба да буде покренута у тим контејнерима. Након тога, менаџери чворова започињу извршавање апликације. Од овог тренутка, даље понашање апликације зависи од самог кода те апликације. [12]

Јарн има одговорност да омогући правилно извршавање апликацијама које се извршавају на ХДФС-у па стога мора на неки начин обрадити грешке које се могу десити у кластеру. На пример, за очекивати је да једна од машина у кластеру престане се радом и тако постане неупотребљива. Када се то деси, менаџер ресурса ће менаџер чворова 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 стварао проблем другим апликацијама у систему. [12]

3.9 Апачи Спарк

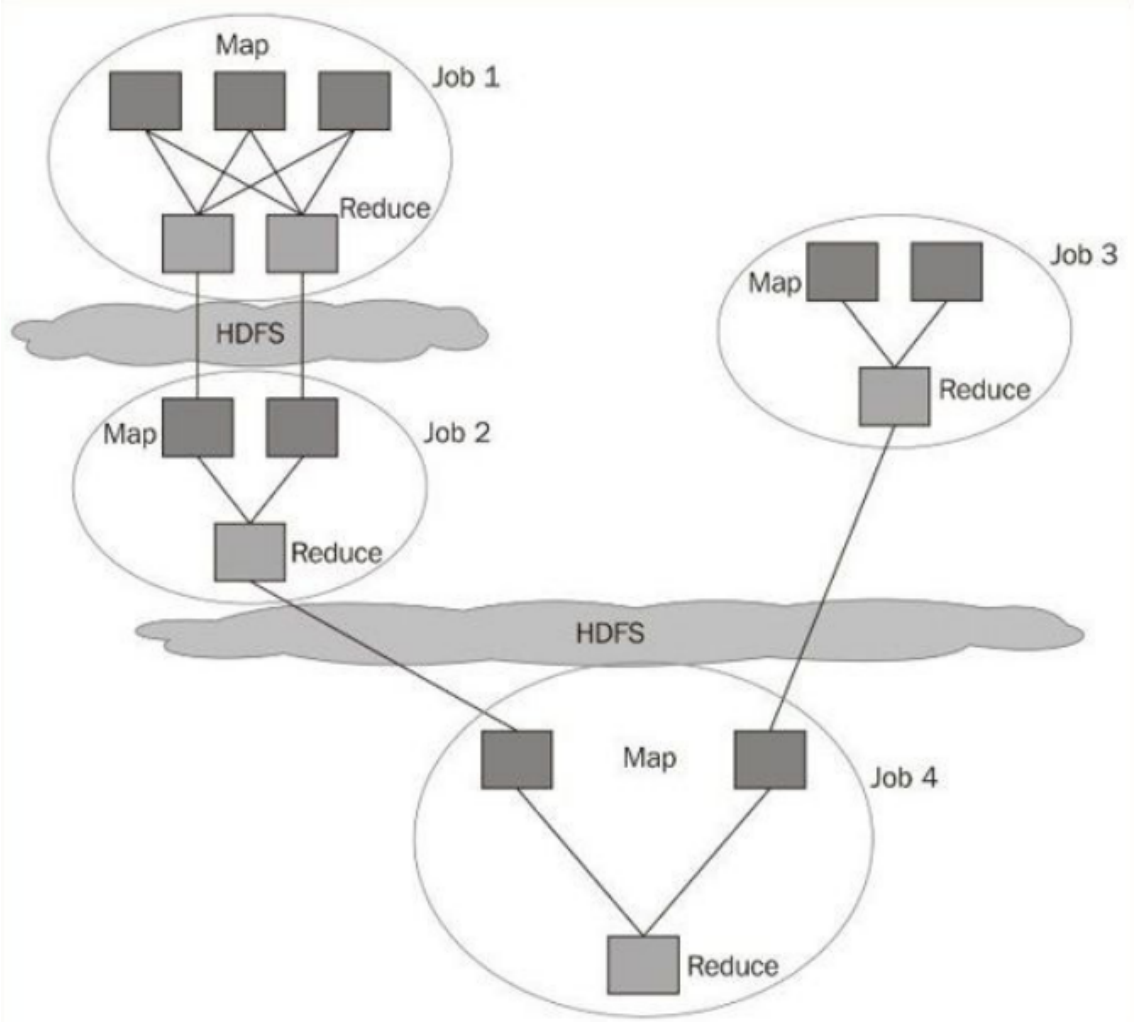
Недостаци Мап Редјуса

Мап Редјус, нажалост, не функционише савршено, што се огледа у проблемима који постоје када су у питању перформансе. Наиме, једна МР апликација је заправо ланац МР послова које треба извршити и сваки од тих послова се састоји из мап и редјус фаза. Излаз једног посла представља улаз у други и тако даље док се не добије коначно решење. Пример једне МР апликације је приказан на слици 3.11.

Међутим, такав приступ има цену а то је да се излаз који генерише један МР посао чува унутар ХДФС-а и одатле ће му приступити остали МР послови којима је тај излаз потребан. Другим речима, међурезултати послова се чувају на диску, што ствара додатне И/О операције и тиме успорава извршавање целокупне апликације. Поред тога, не постоји начин да се ти МР послови на неки начин оптимизују као на пример неком комбинацијом или слично, већ су независни. Да би се решили ови проблеми направљен је нови алат који је данас скоро потпуно избацио МР из употребе, Апачи Спарк (енг. *Spark*). [12]

Увод у Апачи Спарк

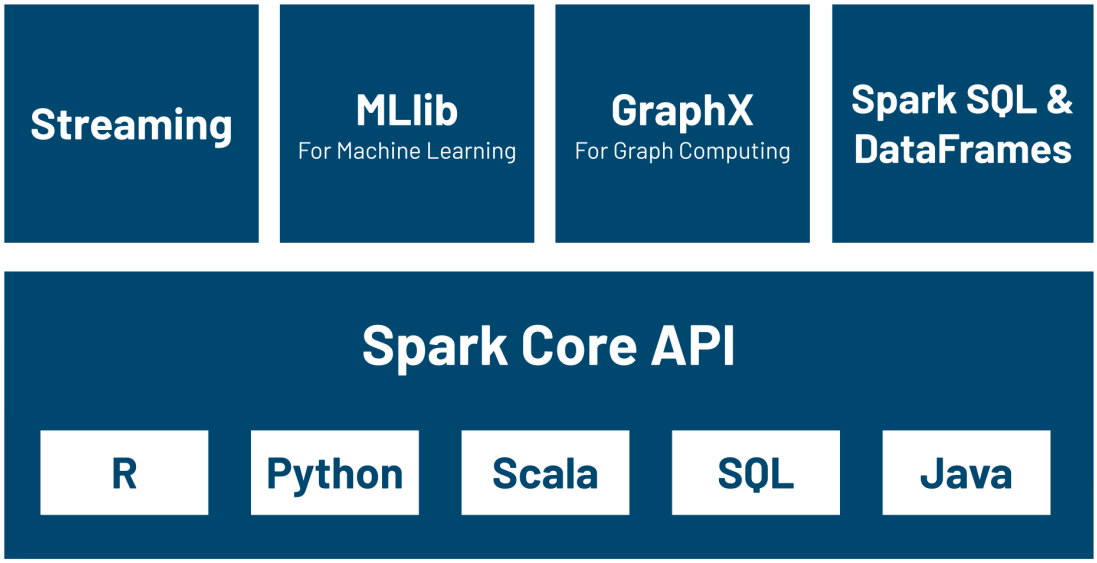
Апачи Спарк је скуп библиотека за паралелну обраду података на кластерима које се користе за рад за подацима, а не за њихово складиштење. Подржава неке од најкоришћенијих програмских језика данашњице, Пајтон (енг. *Python*), Јаву (енг. *Java*), Скалу (енг. *Scala*) и Р (енг. *R*) и може се



Слика 3.11: пример ланца мап редјус послова

користити за разне врсте проблема, од обраде података, одраде токова података, машинског учења и рада са графовима. Лако се скалира и може се користити како на једном рачунару, тако и на кластеру. На слици 3.12 су приказане компоненте Апачи Спарка од којих ћемо се са неким упознати у овој секцији. [1]

Настао је 2009. године на универзитету Беркли у Калифорнији. Написан је у програмском језику Скала и конструисан је са идејом да користи концепте функционалног програмирања. Постао је део Апачи фондације 2013. године и од тада је доживео велики број промена. До сада, избачене су три верзије Апачи Спарка, редом, Спарк 1.0 (2013. године), Спарк 2.0 (2016. године) и Спарк 3.0 (2020. године).



Слика 3.12: Компоненте Спарка

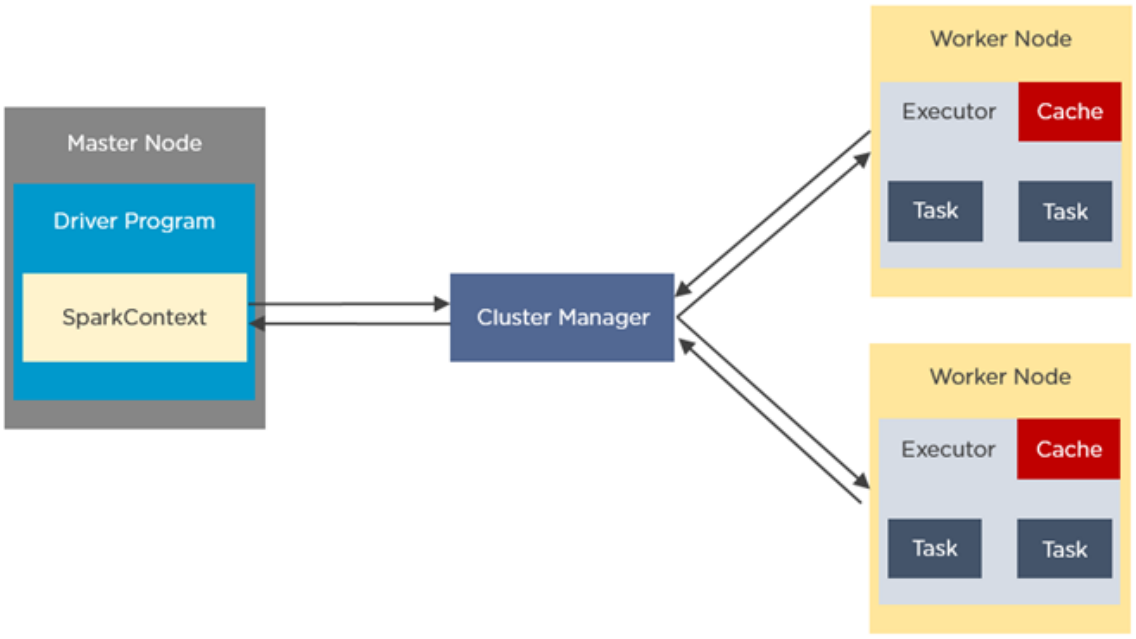
Архитектура

Да бисмо користили Спарк на кластеру, прво је потребно да се прикачимо на њега коришћењем ресурс менаџера. Иако Спарк има свој лични ресурс менаџер, можемо користити и већ поменути Јарн или неки други. Након тога, можемо покретати Спарк апликације на кластеру.

Свака Спарк апликација се састоји из једног драјвер процеса (енг. *driver process*) и једног или више егзекјутор процеса (енг. *executor process*). Драјвер процес је срце Спарк апликације и задужен је за три ствари: одржавање информација о апликацији, реаговање на програм апликације и њен унос као и анализирање, распоређивање и планирање послова који се извршавају на егзекјуторима.

Егзекјутори су ти који извршавају неки посао који им драјвер програм задаје. Сваки егзекјутор је задужен за две ствари: извршавање кода који му драјвер додељује и пријављивање стања извршавања тог кода драјверу. Једноставни приказ Спарк апликације је приказан на слици 3.13. [1]

На приказаној слици се појављује концепт који није поменут до сада, а то је спарк контекст (енг. *spark context*). Налази се унутар драјвера и његова улога је да дефинише конекцију ка кластеру. Поред тога, користи се за дефинисање спаркових апстракција које називамо РДД, о којима ће бити речи у наставку.



Слика 3.13: Архитектура Спарка

Напоменимо да иста архитектура важи и када спарк покрећемо у локалном моду, на једној машини. Тада ће драјвер и егзекјутори и даље бити раздвојени процеси али ће радити на истој машини, а не на засебним. [1]

Партиције

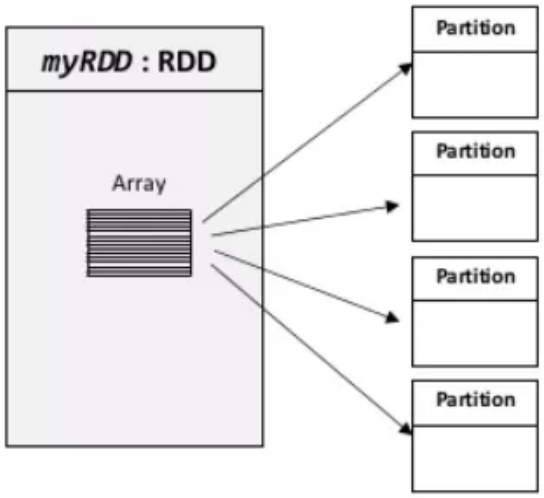
Да би егзекјутори могли паралелно да извршавају операције над подацима, Спарк их дели на делове које називамо партиције (енг. *partitions*). Партиција је дакле један део колекције података који се налази на једној машини кластера. [1]

Уколико су подаци партиционисани само једном партицијом сви ће се бити обрађени од стране једне машине у кластеру, независно од тога колико машина постоји. Слично, уколико смо креирали више партиција, али имамо само један егзекјутор, ниво паралелизма ће бити један због тога што постоји само једна машина која може да обради те податке. [1]

РДД

Основна јединица рада у Спарку се назива РДД (енг. *RDD, resilient distributed dataset*) и све операције са подацима се извршавају преко ње. РДД

је колекција елемената за које важи да су партиционисани по машинама кластера и да се над њима паралелно могу извршавати операције. [9] Дакле, један РДД се састоји од једне или више партиције падатака. [1]



Слика 3.14: Спарк РДД

Постоји неколико начина преко којих можемо направити нови РДД:

- Читањем неког фајла који се налази на фајл систему (обично ХДФС)
- Од неке колекција података програмског језика у коме користимо спарк. Тај процес дељења колекције у партиције називамо паралелизација
- Од неког већ постојећег РДД
- Кеширањем постојећег РДД

Данас се РДД апстракција не користи директно, већ постоје друге које су конструисане над РДД апстракцијом које су је потиснуле, па се РДД сматра застарелим.

Трансформације

Спарк је конструисан по принципима функционалног програмирања па су све његове структуре података имутабилне, што значи да се након њиховог креирања не могу мењати. Ако податке не можемо мењати, како их онда можемо обрађивати? Спарк то решава тако што свака промена над подацима

у ствари креира потпуно нову струкуру података. На пример, уколико имамо један РДД и на неки начин желимо да променимо податке које он садржи, нећемо променити тај РДД, већ ћемо од њега направити нови који у себи садржи измењене податке. [1]

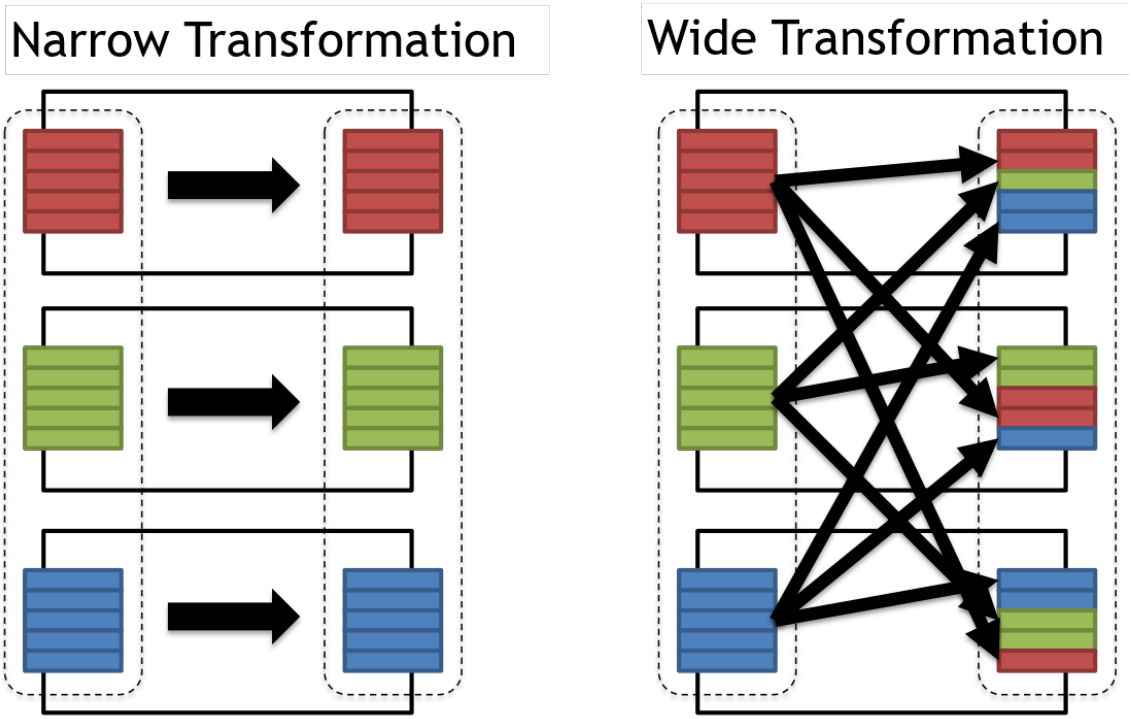
Тај процес, где од једног РДД-ија применом неких наредби добијамо други, називамо трансформација (енг. *transformation*). Пратећи функционалне концепте, трансформације немају никакве бочне ефекте, што значи да се од једног РДД-ија применом истих трансформација увек као резултат добија један исти РДД,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

Постоје две различите врсте трансформација, уске (енг. *narrow*) и широке (енг. *wide*). За уске трансформације важи да једна партиција у оригиналном РДД доприноси настајању највише једне партиције у зависном РДД. Са друге стране, широке трансформације су такве где једна партиција почетног РДД учествује у конструисању више партиција зависног РДД, често свакој. Ово се другачије назива и мешање (енг. *shuffle*) због тога што се врши размена партиција по кластеру. Јасније објашњење трансформација је приказано на слици 3.15. [1]

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

Лења евалуација

Лења евалуација значи да ће Спарк сачекати последњи тренутак да изврши дефинисане трансформације. Када имамо РДД и над њим дефинишемо неколико трансформација, оне неће бити одмах извршене, већ ће се од њих направити план трансформација који ће се извршити тек када се затражи њихов резултат. Разлог тога је ефикасност. Када Спарк зна које ће се све трансформације извршити над неком структуром података, може оптимизовати цео процес на такав начин да добијање резултата траје најкраће могуће. [1]



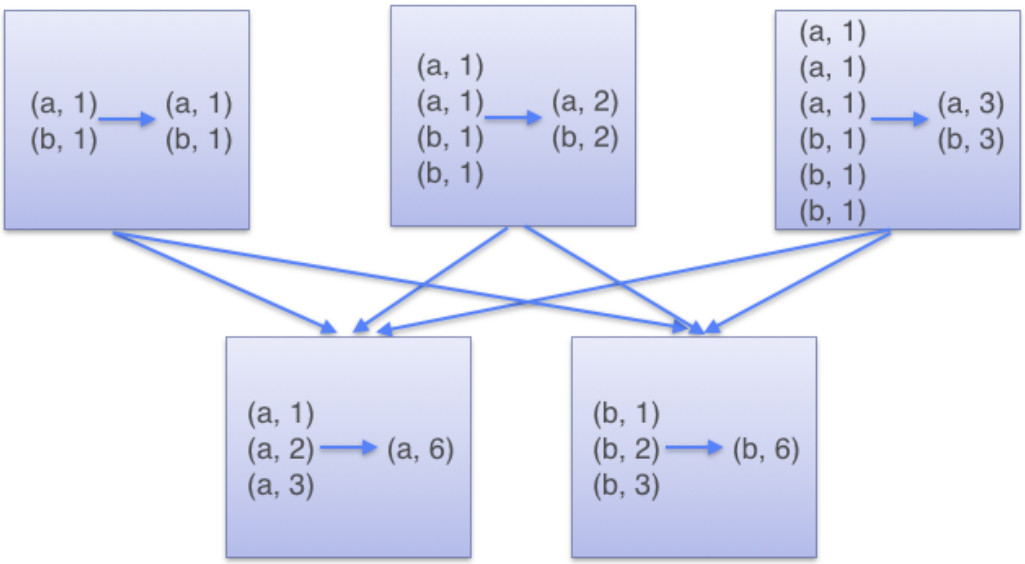
Слика 3.15: Приказ трансформација Спарка

Врсте РДД трансформација

У овој секцији ћемо се упознати са неким трансформацијама које се најчешће користе. Вероватно најпознатија трансформација је *map()* трансформација која за сваки елемент почетног скупа података производи нови, примењујући неку операцију на њега. Поред тог мапирања спарк нуди и флет мап *FlatMap()* трансформацију, која за сваки елемент података производи нула, један или више елемената.

Још једна трансформација која се често користи је *filter()* која за сваки елемент почетног скупа производи нула или један елемент новог скупа. Не врши никакве промене елемената, већ само од једног скупа елемената прави други скуп елемената од којих сваки задовољава неки услов.

Од широких трансформација су вероватно најпознатије редуковање по кључу *reduceByKey()* и *join()*. Обе раде са паровима кључ-вредност. Првом је улога да за сваки кључ сабере вредности са којима је упарен (слика 3.16) док други на основу вредности кључа спаја два скупа у један, где ће у резултујућим подацима вредност кључа бити унија вредности кључа два скупа која учествују у спајању.



Слика 3.16: Редуковање по кључу

Поред ових трансформација постоје и *union()*, *intersect()*, *sortByKey()*, *groupByKey()* али и многе друге. [9]

Акције

Спарк акције користимо када желимо да затражимо резултат ланца трансформација. Дакле, тек када над нечим позовемо акцију заправо започињемо извршавање. Добијени резултат се враћа драјвер програму. [1]

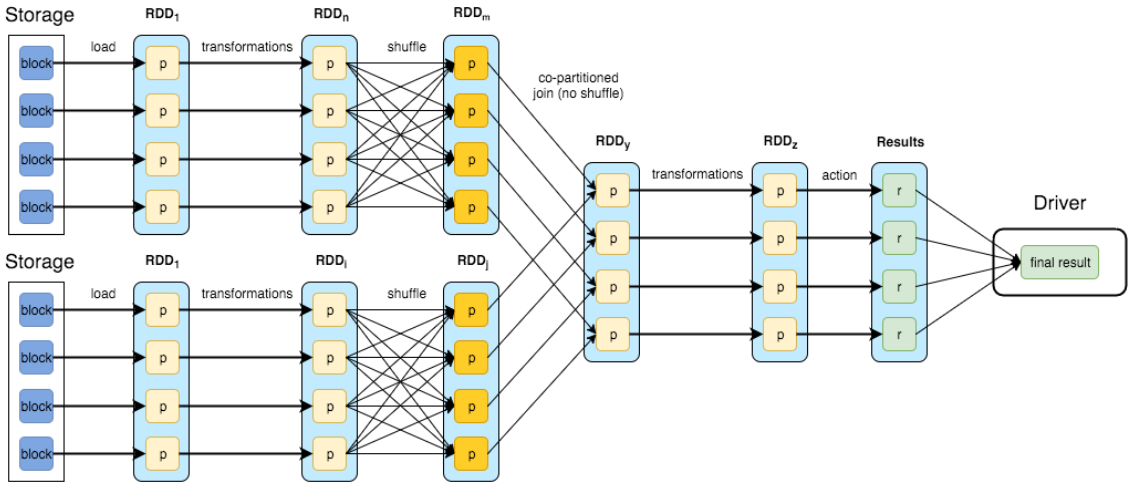
Постоје три врсте акција:

- акције које приказују резултат у конзоли
- акције које исписују резултат у некакав излаз, на пример фајл
- акције које пребацују податке у објекте неког програмског језика у коме користимо спарк

Неке од трансформација су *count()*, *take()*, *saveAsTextFile()*, *collect()* али постоје и друге. [9]

Руковање грешкама у Спарку

Нажалост, могуће је да се током извршавања догоде неки проблеми и да због тога изгубимо партицију. Уколико се то деси, Спарк је у могућности да реконструише изгубљене партиције уз помоћ механизма који се назива граф наследства (енг. *lineage graph*). За сваки РДД, Спарк чува информације о томе од ког је РДД настао и које су трансформације примењене да бисмо добили тај РДД. Тај граф се може приказати сликом 3.17.



Слика 3.17: Пример једног Спарк извршавања

Уз помоћ овог механизма Спарк зна од које партиције је настала свака од партиција и уколико нека партиција нестане, може је рекреирати. То ради тако што се креће кроз граф. Ако једна партиција није валидна, Спарк ће проверити партицију (или партиције) од којих је она настала. Уколико оне постоје, рекреираће невалидну партицију. У супротном, прегледаће рекурзивно партиције од којих је та партиција настала. 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. [12]

Цео овај систем је поуздан из два разлога. Први је тај што трансформације немају бочне ефекте, па знамо да ћемо рекреирањем увек добити један исти РДД. Други је тај што се подаци чувају у ХДФС-у, који је сам по себи поуздан, па ако дођемо у ситуацију да морамо поново да рекреирамо цео ланац из почетка, знамо да ћемо прочитати непромењену вредност са диска.

Кеширање

Веома битна карактеристика Спарка је могућност да чува податке у меморији, односно да их кешира. Када се РДД 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 које користе тај РДД. Овакав приступ знатно побољшава перформансе програма. [9]

РДД можемо кеширати коришћењем *cache()* или *persist()* метода. Када први пут тај РДД буде учествовао у некој акцији, сачуваће се у меморији и моћи ће да се користи у будућности. Кеширање је отпорно на грешке, тако да ако се једна од партиција изгуби, можемо је рекреирати из претходних трансформација. [9]

Постоје различити нивои кеширања у зависности од тога у којој меморији се чувају партиције. Уколико желимо да то буде меморија, можемо позвати *cache()* метод али можемо и *persist()* методу проследити вредност *MEMORY_ONLY*.

Могуће је да у меморији нема места да би се РДД сачувао па због тога то можемо урадити и на диску. Тада прослеђујемо *DISK_ONLY*. Овај приступ се углавном не саветује из тог разлога што је често брже рекреирати цео ланац трансформација поново, него читати вредности са диска. Вредности *MEMORY_ONLY_2* и *DISC_ONLY_2* користимо када желимо да уз чување на тренутној машини извршимо репликацију на још једну машину. Поред тога, можемо бирати мешовити приступ (*MEMORY_AND_DISC*). У том случају ће се РДД кеширати у меморији уколико има простора, а уколико не, чуваће се на диску. [9]

Спарк Дата Фрејм

Дата фрејм 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, иако можемо користити недостајуће вредности. Поред тога, свака колона има један тип и тог типа морају бити све вредности које се налазе у редовима. [1]

Сваки Спарк дата фрејм садржи метаподатке који описују имена колона и њихове типове. Те метаподатке називамо шема (енг. *schema*). Шему можемо дефинисати мануелно али је и можемо закључити из података. [1]

Спарк нуди велики број типова колона, који се мапирају у типове програмских језика преко којих користимо Спарк. Постоје једноставни типови попут целобројних и децималних бројева и ниски али постоје и сложени попут низова, мапа и датума. [1]

Трансформације и акције дата фрејма

Све што важи за трансформације и акције које примењујемо над РДД, важи и за трансформације и акције које примењујемо над дата фрејмом. Трансформације немају бочни ефекат и лење су, тако да се извршавају тек када се над њима позове нека акција.

Једина разлика је та што РДД и дата фрејм другачије представљају податке па су им акције другачије. На пример, дата фрејм има акцију *select(columns)* коју користимо да од једног дата фрејма добијемо други али са подскупом колона првог. Друга је *filter(condition)* коју користимо да бисмо изабрали одређене редове једног дата фрејма и тако направили други. Ове две трансформације су идентичне редом *SELECT* и *WHERE* наредбама у програмском језику *SQL*. Поред њих постоји и *withColumn(name, expression)*, уз помоћ које додајемо нову колону на постојећи дата фрејм. Наравно, постоји још велики број других. [1]

Неке од акција које можемо применити на дата фрејм су на пример, *collect()*, која нам омогућава да користимо податке унутар дата фрејма у програмском језику преко ког користимо Спарк и *show()* коју користимо када хоћемо да испишемо дата фрејм на стандардни излаз. [1]

Разлика између дата фрејма и РДД

Поставља се питање зашто бисмо користили један од ова два модела пре другог. РДД се користи за програмирање ниског нивоа и омогућава нам да директно вршимо операције над партицијама. Уколико желимо да користимо РДД за обраду података морамо бити веома пажљиви како пишемо код, коју трансформацију примењујемо и када, због тога што то може бити веома битно када су у питању перформансе. [1]

Са друге стране, дата фрејм нема тих проблема. Све трансформације које напишемо над неким дата фрејмом пролазе кроз процес оптимизације тако

да ћемо увек добити најоптимизованији могући код. Оптимизатор који је задужен за овај процес називамо Каталист (енг. *Catalyst*). [1]

Због тога, али и због једноставнијег интерфејса, дата фрејм је скоро потпуно потиснуо РДД из употребе.

Планови извршавања

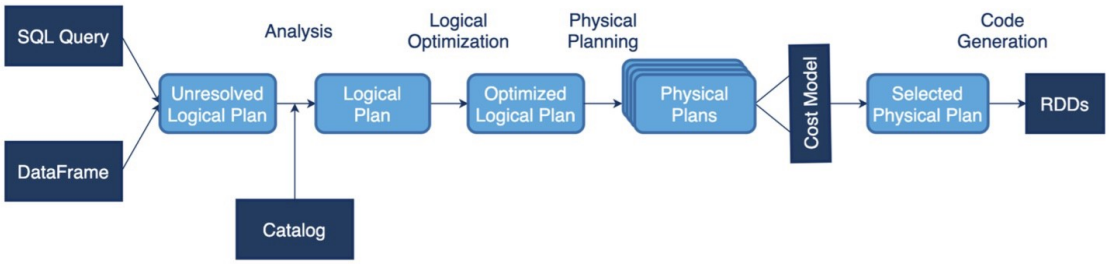
Процес извршавања дата фрејм кода изгледа овако:

1. Напишемо дата фрејм код
2. уколико је исправан, од њега се прави логички план
3. Спарк од логичког плана конструише физички план, успут примењујући оптимизације
4. Добијени физички план се пребацује у РДД и извршава се

Дакле, након што се дата фрејм код прогласи исправним, од њега се конструише неразрешен логички план (енг. *unresolved logical plan*) који представља апстрактне трансформације које треба извршити, али не садржи никакве изформације о томе над којим табелама или колонама треба извршити те трансформације. Те информације Спарк добија уз помоћ каталога (енг. *catalog*) који садржи потребне информације. Резултат примене каталога на неразрешен логички план је логички план (енг. *logical plan*). Након тога се тај план оптимизује од стране Каталиста, који оптимизује логички план тако што га анализира и примењује одређена правила на њега. Резултат је оптимизовани логички план (енг. *optimized logical plan*). [1]

Након што се оптимизовани логички план успешно креира, Спарк у односу на њега конструише неколико физичких планова (енг. *physical plan*). Физички план дефинише како ће се логички план извршити на кластеру. Сви физички планови се након тога евалуирају и бира се онај који даје најбољи резултат. Тај физички план се онда преводи у РДД трансформације и ивршава на кластеру. [1]

Цео процес је приказан на слици 3.18.



Слика 3.18: Ток извршавања Спарк дата фрејма

Остале компоненте Спарка

Поред поменутих компоненти које нам Спарк нуди постоје и многе друге али се њима нећемо бавити у овом раду због тога што су ван опсега. Спарк, поред РДД и дата фрејм интерфејса нуди још два, дата сет (енг. *spark dataset*) и спарк сикуел (енг. *spark sql*). Веома су слични дата фрејму и пролазе кроз исти процес компилације али ипак постоје неке мање разлике. [1]

Спарк сикуел се користи за извршавање упита над кластером. Једина разлика је у томе што се грешке које настају овим методом појављују док се код извршава, а не у компилацији. [1]

Разлика између дата сета и дата фрејма је у типовима. Наиме, разлика је у томе што што се типови колона дата фрејма проверавају са онима у шеми у току извршавања програма, док се код дата сета то дешава за време компилације. Такође, разлика је у томе што је дата сет доступан само у JBM (енг. *JVM*) базираним језицима, скали и јави, док у другима не постоји. [1]

Поред тога Спарк омогућава конструисање модела машинског учења над подацима преко библиотеке која се назива *Spark ML lib*. Она се може користити за препроцесирање, тренирање модела и прављење предвиђања. [1]

Спарк нуди и операције над стримовима (токовима) података. Можемо се закачити за некакав ток и над њим можемо примењивати исте операције које смо поменули у претходним секцијама овог рада (на пример у дата фрејм секцији). Спарк стриминг омогућава преплату на токове података који настају из неког веб сокета (енг. *socket*), од неке групе фајлова, или које производи Апачи Кафка. [10]

За сам крај, поменимо да Спарк поред свега тога има библиотеку за рад са графовима, *graphX* али се данас за обраду графова често користе нека друга решења. [1]

Глава 4

Семантички веб

Веб је током свог постојања константно еволуирао. Разликују се три веће етапе које су назване по редним бројевима, Веб 1.0, Веб 2.0 и Веб 3.0.

4.1 Веб 1.0

Ова верзија представља прву фазу еволуције веба. Настао је у последњим деценијама 20. века и трајао је до средине прве деценије 21. Карактерише га скуп статичког садржаја који је повезан преко веза названих *hyperlik*.

Уз помоћ њега су представљане само информације које су се приказивале корисницима. Још увек није постојао језик за улепшавање страница, *CSS* и није било динамичких линкова, логовања корисника и остављања коментара.

4.2 Веб 2.0

Друга фаза је почела одмах након прве и траје и дан данас. Акценат је стављен на садржај генерисан од стране корисника, једноставно коришћење као и на размени информација. У овој фази је, преко програмских језика који се извршавају на серверској страни, омогућен рад разних апликација попут Амазона (енг. *Amazon*), Ју Тјуба (енг. *YouTube*) и Фејсбука (енг. *Facebook*). Све то је омогућило корисницима да на једноставан начин деле и размењују своја мишљења и искуства.

Међутим, Веб 2.0 има једну великуману, а то је да се сви подаци, које поменуте апликације сакупљају, складиште на приватним серверима које контролишу корпорације. Велики број њих користи те податке да би кориснике

што дуже задржало на својим сервисима. Такође, у неколико инстанци су ти подаци продавани некој другој страни, у циљу једноставне зараде. Због овога се може рећи да су корисници постали производ друге фазе веба.

4.3 Веб 3.0

Циљ треће верзије веба је да се ослободи централизације присутне у вебу 2.0 и направи децентрализован али сигуран интернет где би људи могли да размењују информације без посредника, ког тренутно представља корпорација чије услуге ти корисници користе.

Последица децентрализације је та да апликације више неће бити покретане преко огромних, приватних, база података и сервера, већ ће се прећи на технологије као што су блокчеин (енг. *blockchain*) и мреже равноправних корисника (енг. *peer-to-peer*).

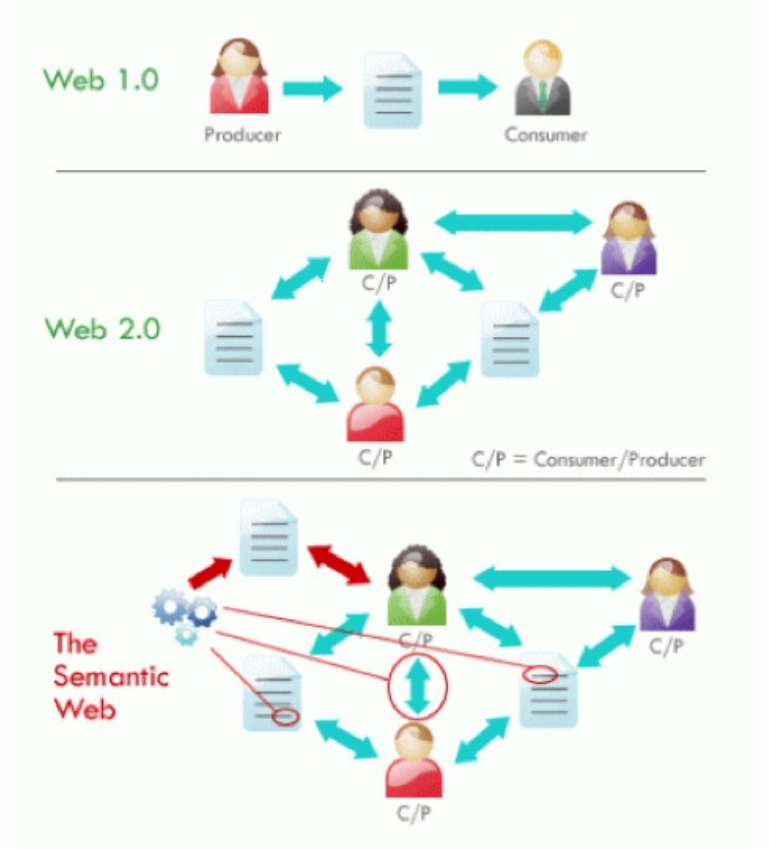
Идеја је да у самом срцу веба 3.0 стоји семантички веб, чија би улога била да повеже постојеће податке и информације које се налазе на интернету, и направи их таквим да њихов садржај и контекст могу разумети машине.

Ова верзија Веба је још увек у развоју, са огромним простором за даље истраживање. Међутим, како је сама архитектура доста комплекснија од архитектуре Веба 2.0, није сигурно да ли ће заживети и ако да, када. Али оно што је сигурно је да идеја о дељеним подацима и децентализованом вебу и даље постоји и велики број људи је развија и унапређује.

4.4 Семантички веб

Семантички веб је надоградња садашњег веба, *WWW*-а, који омогућава рачунарима да, на интелигентан начин, претражују и обрађују податке са веба. Интелигентно у овом случају значи да ће рачунари бити способни да закључе шта подаци представљају људима који их користе. Пошто до сада није конструисан ниједан систем вештачке интелигенције који може потпуно да опонаша човека, то се може постићи само ако се значење, то јест семантика, ресурса на вебу експлицитно представи рачунарима у формату у којем их они могу обрадити. [7]

Да би се ово постигло, није довољно само складиштити податке у језику који машине разумеју, на пример у *HTML* формату, већ је неопходно да се уз



Слика 4.1: Фазе Веба

те податке додају и информације о семантици која јасно говори које закључке треба из њих извући. Међутим, тако нешто је врло вероватно немогуће извести због тога што је чак и људима тешко да се договоре око значења садржаја одређених страница на вебу, па је стога још теже формализовати тај садржај на такав начин да буде значајан машинама. [7]

Да би семантички веб функционисао, на неки начин се људско знање мора изразити неким формалним језиком. Тај проблем се може решити моделовањем. Један од језика који се користи у семантичком вебу је *OWL 2* и настао је под утицајем коришћења моделовања у биологији. На који начин ће знање бити моделовано зависи од тога за шта ће конструисан модел бити коришћен. Код семантичког веба, идеја је да компјутерски програми закључују на основу датих информација на такав начин да узимају у обзир резоновање и формалну репрезентацију знања. [7]

Дакле, развој семантичког веба се заснива на припајању моделовања знања и аутоматског закључивања вебу. Исто тако, на овај начин се веб апликације

уводе у домен формалног моделовања и репрезентације знања.

Иако су раније постојали примери стандардизације саме формалне репрезентације знања, семантички веб је допринео њеној важности и употребљивости. Велика већина релевантних покушаја стандардизације је спроведена од стране *WWW* Конзорцијума, познатијег као *W3C*. [7]

Историјат

Идеја додељивања семантике вебу није нова и постојала је још у самом зачећу веба. Први покушаји су забележени код типизираних линкова (енг. *typed links*) чија је улога била да, поред тога што су представљали референцу ка другом документу, садрже и неке информације о томе шта заправо тај линк представља. [7]

Семантички веб је добио већу пажњу јавности 2001. године, када је Тим Бернерс Ли (енг. *Tim Berners-Lee*) објавио чланак назван *Семантички веб* у којем су представљене идеје о томе како би семантички веб функционисао. До данас је *W3C*, чији је Тим Бернерс Ли члан, објавио неколико технологија међу којима су *RDF, Resource Description Framework*, *OWL, Web Ontology Language* и језик за упите *SPARKQL*, али и многе друге. [7]

Поред тога су забележени и покушаји у томе да се у *HTML* документе, преко вредности атрибута, доделе неки семантички подаци једном веб документу. Те вредности се називају микроформати (енг. *microformats*) и користе се за решавање проблема у специфичним доменима апликација, на пример, приликом енкодирања личних података. [7]

Једноставан пример микроформата је *HTML* атрибут *rel* који се користи да би се назначило шта се налази на линку коме тај атрибут је додељен. У следећем примеру се користи да би се нагласило да се на приложеном линку налази лиценца.

```
<small>This article is licensed under a
  <a rel="license" href="http://creativecommons.org/
   licenses/by-nc-sa/2.0/">
  Creative Commons Attribution Non-commercial Share-alike
  (By-<abbr>NC</abbr>-<abbr>SA</abbr>) license</a>.
</small>
```

Количина семантичких података која је доступна на вебу се повећава из године у годину и подаци су постали међусобно повезани. Ово се дешава због тога што идентификатори који се користе у језицима семантичког веба прате исте принципе као и адресе класичног веба. Стога, име једног објекта семантичког веба се може интерпретирати као веб адреса. То доводи до појаве повезаних података (енг. *linked data*) која се односи на семантичке податке чији су идентификатори заправо показивачи ка веб адресама на којима се може пронаћи више информација о објекту. [7]

Због тога се користи термин *веб података*, који описује семантички веб као покушај који се примарно фокусира на размену података.

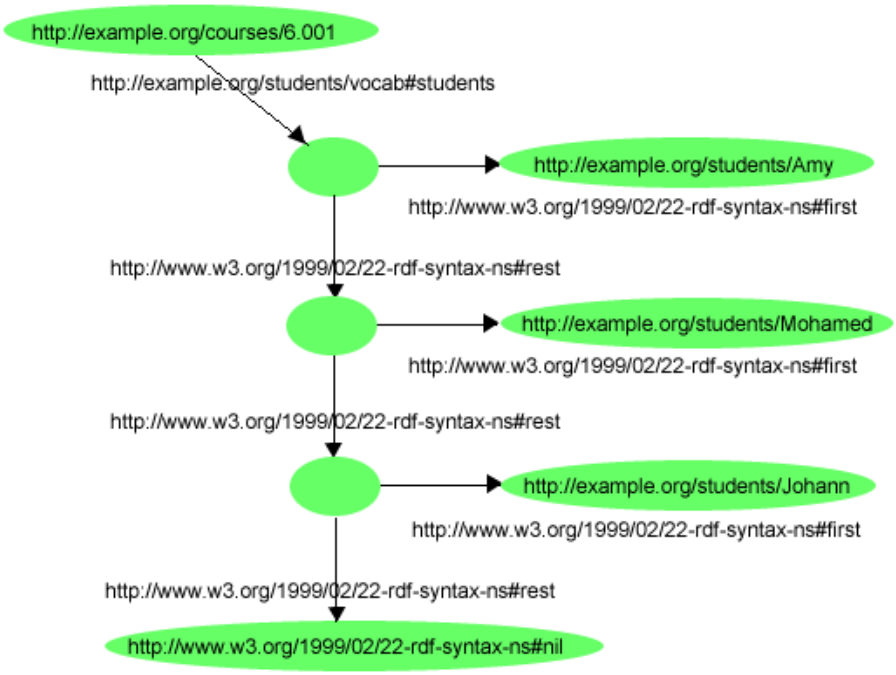
RDF језик

Механизам за описивање ресурса (енг. *resource description framework*) или скраћено *RDF* је формални језик за описивање структуираних информација. Поента овог језика је да омогући апликацијама да размењују податке на вебу задржавајући њихово оригинално значење. За разлику од језика као што су *HTML* и *XML*, циљ није приказати документе у правилном формату већ омогућити обраду информација које они садрже. Због свега наведеног *RDF* се често сматра основом за развој семантичког веба. [7]

Дакле, улога *RDF*-а није да описује структуру докумената, већ да опише однос између неких објеката. Данас је овај формат веома распростањен и готово сваки виши програмски језик поседује библиотеке помоћу којих се могу извршавати разне операције над њим. [7]

RDF документ је описан усмереним графом, односно скупом чворова који су повезани усмереним гранама. Сваки чвор и свака грана имају засебан идентификатор. Структура графа је изабрана баш из тог разлога да би се изразио однос између објеката, а не њихова хијерархија. На пример, на слици 4.2 се примећује да је однос између приказаних елемената, курса и студената, информација која нема јасну хијерахију. *RDF* користи такав однос као основну јединицу информације. Штавише, велики број таквих односа производи графове. [7]

Други разлог због кога је *RDF* представљен графом је чињеница да му је улога да служи као описни језик података веба и других мрежа. Информације у овим срединама се складиште и обрађују дистрибуирано па је стога битно комбиновати *RDF* податке на једноставан начин. На пример, граф који је



Слика 4.2: Пример РДФ графа

репрезентација једних података се лако може повезати са графом који представља нешто друго, једноставним повезивањем. Ако претпоставимо да поред графа на слици 4.2 постоји, на пример, још један граф који описује студенте, лако се може закључити да се они једноставно могу повезати и користити заједно. [7]

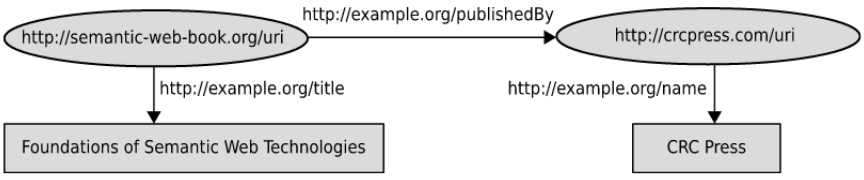
Додељивање имена *RDF* документима може произвести проблеме. Наиме, именовање не мора бити униформно. На пример, могуће је да два документа садрже информације о истој теми, али да су им идентификатори потпуно различити. Такође, могуће је и да је једном истом ресурсу додељено неколико различитих имена или да се исти идентификатори користе за различите појмове.

Да би се тај проблем избегао *RDF* користи униформни индикатор ресурса (енг. *uniform resource identificator, URI*) као имена, у циљу разликовања ресурса. Напоменимо да је *URI* генерализација *URL* адреса које се користе за приступ документима на интернету. Како је сваки *URL* валидан *URI*, може се користити као идентификатор унутар *RDF* докумената. У великом броју апликација циљ није размени информације о веб страницама већ о великом броју објеката као што су на пример студенти, књиге, места, догађаји

и тако даље. Таквим објектима се не може приступити на интернету па се њихов *URI* користи ускључиво за идентификацију. Напоменимо и то да се *URI* адресе које нису *URL* називају *URN*. [7]

Због идентификације, чворовима и гранама унутар *RDF* графа се додељује *URI* као име, што можемо приметити и на већ приложеном примеру (слика 4.2). Ово правило има два изузетка. Први је тај да је ипак могуће доделити име које није *URI*, а други је да је могуће не доделити име и оставити га празним, али се таквим случајевима нећемо бавити. [7]

Иако *URI* представљају имена, њихово значење је подложно интерпретацији па другачији алати могу на другачији начин да посматрају њихово значење. Због тога је уведен још један појам, литерал, који представља вредности унутар *RDF*-а. Увек су записани ниском уз коју је припојен и њен тип. На пример, ниска *„123"* може бити целобројног типа што би значило да представља број *123*. За разлику од *URI*-ја, литерали увек имају једно значење. У графу се представљају правоугаоним чворвима (слика **??**)). [7]



Слика 4.3: Пример РДФ графа са литералима

Репрезентација

Како су *RDF* графови ретки, представљају се скупом грана које се у њему налазе. Једна грана је представљена вредношћу чворова, као и њеном ознаком. Чворове називамо субјекат и објекат, док се ознака назива предикатом. Тројка субјекат-предикат-објекат се другачије назива и *RDF* тројком. Сваки члан тројке може бити некакав *URI* али може бити и литерал. [7]

Један од формата који се користе за репрезентовање *RDF* графа је корњача (енг. *turtle*) нотација. У овој нотацији се граф са слике 4.3 приказује на следећи начин:

```
<http://semantic-web-book.org/uri> <http://example.org/
    publishedBy> <http://crcpress.com/uri> .
```

```
<http://semantic-web-book.org/uri> <http://example.org/
    title> "Foundations of Semantic Web Technologies" .

<http://crcpress.com/uri> <http://example.org/name> "CRC
    Press"
```

Међутим, овај запис, тачније *URI* који му припадају, се може скратити увођењем префикса који означавају именска поља која представљају фамилију *URI* адреса. Уз помоћ њих се исти запис може записати доста једноставније:

```
@prefix book: <http://semantic-web-book.org/> .
@prefix ex: <http://example.org/> .
@prefix crc: <http://crcpress.com/> .

book:uri ex:publishedBy crc:uri .
book:uri ex:title "Foundations of Semantic Web Technologies
    " .
crc:uri ex:name "CRC Press" .
```

Поред корњача нотације, чест начин репрезентовања *RDF* структуре је уз помоћ *XML* формата. Разлог томе је што је тај формат читљив и људима и машинама, али и тај што скоро сваки програмиски језик поседује алате за обраду тог формата. Приказ једноставног *RDF* графа са слике 4.3 у *XML* формату се може наћи у наставку:

```
?xml version="1.0" encoding="utf-8"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax
    -ns#"xmlns:ex ="http:/example.org/">
  <rdf:Description rdf:about="http://semantic-web-book.org/
    uri">
    <ex:publishedBy>
      <rdf:Description rdf:about="http://crcpress.com/uri">
      </rdf:Description>
    </ex:publishedBy>
  </rdf:Description>
</rdf:RDF>
```

OWL

OWL, скраћено од *Web Ontology Language*, је језик семантичког веба који се користи за представљање комплексног знања о стварима, њиховим групама или њиховим релацијама. Комплексно знање у овом контексту су инфомације које се не могу изразити преко *RDF* формата. Овај језик спада у групу логичких језика, па се знање које изражава може користити унутар компјутерских програма. *OWL* документ се назива онтологијом и могуће га је поставити на интернет, одакле га могу реферисати неке друге *OWL* онтолигије. [7]

Тренутна верзија овог језика је *OWL 2*.

SPARQL

Иако се информације из претходно поменутих *RDF* и *OWL* формата могу сазнати њиховом обрадом, то често у пракси није довољно. Због тога су настали упитни језици, чија је улога извлачење потребних информација из разних структура података семантичког веба. Упитни језик за *RDF* је назван *SPARKQL*. [7]

Спаркл, или *SPARQL*, *SPARQL protocol and RDF query language*, је стандард за упите информација *RDF* формата као и за представљање добијених резултата. Иако је синтакса овог језика веома слична синтакси језика *SQL*, користе се за упите над тотално другачијим структурама података. [7]

Пример једног *SPARQL* упита је приказан у наставку.

```
PREFIX ex: <http://example.org/>
SELECT ?title ?author
WHERE {
  ?book   ex:publishedBy   <http://crc-press.com/uri> .
  ?book   ex:title         ?title .
  ?book   ex:author        ?author
}
```

Приказани упит се састоји из три дела. Први део је одређен са речи *PREFIX* и означава именски простор, исто као у корњача нотацији. Након њега следи *SELECT* који одређује формат резултата који ће бити приказан. Трећа фаза, у којој се заправо упит и извршава, је означена са *WHERE* коју прати некакав приказ графа. У приказаном примеру је граф представљен са три тројке које означавају која правила морају да важе за информације које

упитом желимо да добијемо. У овом случају су то наслови и аутори чланака које је објавио *CRC Press*, таквих да наслов и аутор постоје. [7]

Наравно, Спаркл језик је доста моћнији од једноставног примера који је приказан, али улазак у детаље је ван опсега овог рада.

Употреба семантичког веба

Семантички веб је и даље млада технологија која се развија, и тренутно се налази у фази између примене и развоја, али има велики потенцијал и може имати велики број примена. У последње време неке веб странице и портали почињу да користе *RDF* и метаподатке и на тај начин постају део уланчаних података (енг. *linked data*). У последње време се појављују и семантичке википедије, које су сличне као и већ постојеће, с тим што омогућавају кориснику да мења метаподатке који допуњује странице. Поред њих се развијају и семантички портали, веб странице где поред информација доступних човеку, постоје и друге, онтолошке, намењене машинама. Те информације се користе у циљу побољшања искуства корисника који користе те портале. [7]

Глава 5

OSM

5.1 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 5.1: 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]

5.2 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 6

App

6.1 Opis

ovde opis appa i sta se trazi od nas

6.2 Cloud

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

6.3 Arhitektura aplikacije

opis komponenti

6.4 Подаци

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

6.5 Obrada OSM skupa Sparkom

koje spark transformacije su primenjene na koje podatke

6.6 PolyContains

algoritmi korisceni za poly contains i rezultati (graficki prikaz)

6.7 Rezultat

dobijeni rezultati aplikacije, prikaz nekih selekcija itd

Глава 7

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. Hdfs architecture guide. online at: https://hadoop.apache.org/docs/.

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

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

[6] OpenStreetMap. Openstreetmap wiki.

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

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

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

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

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

[12] 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.