УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ

Давид Гавриловић

ДИСТРИБУИРАНА ОБРАДА ГЕОПРОСТОРНИХ ПОДАТАКА

мастер рад

Београд, 2022.

Ментор:

др Милена Вујошевић Јаничић, ванредни професор
Универзитет у Београду, Математички факултет

Чланови комисије:

др Саша Малков, ванредни професор
Универзитет у Београду, Математички факултет

др Мирко Спасић, доцент
Универзитет у Београду, Математички факултет

Датум одбране: септембар 2022.

Наслов мастер рада: Дистрибуирана обрада геопросторних података

Резиме: Дистрибуирани системи су системи који су сачињени од умрежених машина које међусобно сарађују и паралелно извршавају посао. Пример таквог система је *HDFS*, централни део екосистема *Hadoop*. У дистрибуираним системима се обрада података остварује поделом података на делове који се паралелно обрађују на појединачним машинама система. *Apache Spark* је алат за дистрибуирану обраду података. Геопросторни подаци су подаци који представљају локације на географској мапи, заједно са њиховим описом. Циљ рада је конструкција апликације која на дистрибуиран начин обрађује геопросторне податке скупа *OpenStreetMap*. За дистрибуирану обраду података се користи програмски језик Скала и *Apache Spark*.

Кључне речи: програмски језик Скала, дистрибуирана обрада података, *Hadoop*, *Apache Spark*, геопросторни подаци, *OpenStreetMap*

Садржај

1 Увод **1**

2 Програмски језик Скала **3**
2.1 Особине језика Скала 3
2.2 Интерпретер за Скалу 6
2.3 Типови 7
2.4 Промењљиве 7
2.5 Контрола тока 8
2.6 Функције 9
2.7 Објектно оријентисана својства језика 10
2.8 Колекције 16

3 Дистрибуирана обрада података **18**
3.1 Доба података 18
3.2 Скалирање система 19
3.3 Организација дистрибуираних система 20
3.4 Систем *Hadoop* 21
3.5 Дистрибуирани фајл систем *HDFS* 22
3.6 Парадигма *MapReduce* 25
3.7 Преговарач ресурса *Apache Yarn* 29
3.8 Остале компоненте *Hadoop*-а 32

4 Алат *Apache Spark* **33**
4.1 Архитектура 33
4.2 Партиције 34
4.3 Апстракција података *RDD* 35
4.4 Апстракција података *DataFrame* 42

4.5 Остале компоненте *Spark*-а 49

5 Скуп података *OpenStreetMap* 51

5.1 Елементи 52
5.2 Чворови 53
5.3 Путање 53
5.4 Релације 54

6 Опис апликације *Geo-locator* 56

6.1 Рачунарство у облаку 56
6.2 Подаци 58
6.3 Архитектура апликације 58
6.4 Одређивање припадности локације држави 59
6.5 Обрада геопросторних података *Spark*-ом 61
6.6 Сервер апликације 73
6.7 Клијент апликације 75
6.8 Приказ резултата 76

7 Закључак 79

Библиографија 81

Глава 1

Увод

Данас се, због развоја интернета, друштвених мрежа и сличног, генерише велика количина података. Обрада тих података је битна јер се из њих могу добити разне корисне информације. Због бољих перформанси у односу на појединачне машине, за обраду великих количина података се користе дистрибуирани системи, који су у могућности да поделе податке на делове и да те делове паралелно обрађују на појединачним машинама.

Један од примера велике количине података су геопросторни подаци. То су подаци који представљају локације на географској мапи и њихов опис. Могу се користити за конструкцију географских мапа, лоцирање разних објеката на мапи, одређивање оптималних линија градског превоза и слично. Пример јавно доступног скупа геопросторних података је *OpenStreetMap* који садржи локације одређене географском дужином и ширином као и информације о томе шта се на локацијама налази (аутобуска станица, пешачки прелаз, црква, ресторан и слично).

Циљ рада је израда апликације *Geo-locator*. Њена функција је да филтрира геопросторне податке и да за сваку локацију која представља битан туристички податак (хотел, хостел, ресторан, паб, болницу и слично) одреди којој држави припада. Филтрирани подаци се касније приказују на географској мапи света. Коришћени подаци припадају скупу *OpenStreetMap* и обрађују се на дистрибуиран начин коришћењем програмског језика Скала и алата *Apache Spark*.

Апликација је израђена у програмском језику Скала, па су у поглављу 2 описани њени основни концепти. Такође, Скала је заступљена и у алатима за дистрибуирану обраду података. У поглављу 3 је приказан концепт дистрибу-

ираних система као и мотивација за њихово постојање. У истом поглављу су описани структура, начин рада и компоненте дистрибуираног система *HDFS*. Приказан је и начин рада, али и недостаци *MapReduce*-а, прве парадигме за дистрибуирану обраду података. Опис алата *Apache Spark* и приказ његовог начина функционисања, архитектуре и компоненти се налази у поглављу 4, док се детаљнији опис скупа *OpenStreetMap* и његових елемената налази у поглављу 5. Опис апликације, њене компоненте и технологије које се користе у њеној изради су приказане у поглављу 6. У последњем поглављу се налази закључак, у коме се налазе коментари о раду и апликацији, као и могућа унапређења.

Глава 2

Програмски језик Скала

Скала (енг. *Scala*) је виши програмски језик заснован на функционалној и објектно оријентисаној парадигми [32]. 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.

Настала је 2001. године 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 распростањена и веома је заступљена у заједници отвореног кода у пројектима као што су *Apache Spark* [21], *Apache Kafka* [19], *Apache Flink* [16] и *Akka* [30].

2.1 Особине језика Скала

Скала 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 [32]. 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*.

Поред тога што је објектно оријентисан језик, Скала је и функционалан језик [32]. Функционално програмирање је засновано на двама принципима. Први је да су функције вредности првог реда. То значи да се функције посматрају на исти начин као и други типови, на пример целобројни тип или тип ниске. Такође, функције је могуће прослеђивати другим функцијама, функције могу бити повратна вредност неке друге функције и функције се могу складиштити у променљивама.

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

Из овога произлази да функционални језици користе непроменљиве структуре података [32]. 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.

Међутим, иако подржава писање функционалног кода, Скала није чисто функционалан језик, што значи да је ипак могуће дефинисати функције које поседују бочне ефекте и да је могуће користити структуре података које се могу мењати. Ипак, у Скали се писање кода који није функционалан не препоручује и увек се тежи поштовању концепата функционалне парадигме.

Повезаност са језиком Јава

Скала се компајлира у Јавин JBM бајткôд (енг. *Java JVM bytecode*) [32]. То значи да Скала 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. Време извршавања Скала програма приближно је једнако времену извршавања Јава програма.

Међутим, иако се компајлирају на исти начин, програми написани у језику Скала често садрже мањи број линија од оних написаних у језику Јава. У

неким случајевима се очекује да је кôд чак дупло краћи. Краћи програми доводе до тога да је кôд лакше писати и разумети, али и до мање вероватноће прављења грешака.

Један од многих примера како Скала смањује број линија у односу на Јаву је приказан у кодовима 2.1 и 2.2 који представљају начине декларисања класе у та два програмска језика. У Скали се не мора декларисати конструктор, што доводи до смањења броја линија кода.

```
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

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

Кôд 2.2: Декларација класе у језику Скала

Статичка типизираност

Статичка типизираност значи да се типови промењивих закључују за време компајлирања програма. Супротан термин је динамичка типизираност, која закључује типове за време извршавања. Оба приступа имају своје предности и мане. Скала је статички типизиран језик и поседује веома напредан систем типова.

Статичка типизираност доноси предности које доводе до лакшег откривања грешака приликом писања кода [32]. На пример, у статички типизираним програмима се током компајлирања сазнаје да ли је примењена нека операција на објекат типа 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.

Скала није само статички типизиран језик већ је и језик који аутоматски закључује типове у току компајлирања. На пример, када се декларише нека

променљива, нема увек потребе назначити и њен тип, пошто га компајлер често може аутоматски одредити. То значи да се следеће две линије кода (пример 2.3) понашају еквивалентно.

```
// primer 1
val x: Int = 10
// primer 2
val x = 10
```

Кôд 2.3: Декларација променљиве са и без експлицитног навођења типа

Скала програмер не мора експлицитно да наводи типове, али је то често пожељно. Навођење типова осигурава да ће кôд заправо користити тип који му је намењен. Такође, навођење типова побољшава читљивост програма и представља вид документације.

2.2 Интерпретер за Скалу

Скала је језик који се може интерпретирати. Да би се покренуо интерпретер за Скалу потребно је покренути команду *scala* (кôд 2.4).

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

scala>
```

Кôд 2.4: Интерпретер за Скалу

Након што се унесе кôд у интерпретер и притисне ентер, покреће се интерпретација написаног кода и излаз се приказује у конзоли. Пример извршавања кода у интерпретеру је приказан у коду 2.5.

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

Кôд 2.5: Пример извршавања кода у интерпретеру

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

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 [32]. На пример, логички тип у Скали, *scala.Boolean* је еквивалент Јавином примитивном типу *boolean*. Исто важи и за друге примитивне типове Јаве попут целобројног (*Int*) и типова са покретним зарезом (*Float* и *Double*).

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

Сваки тип, долази са скупом оператора који се могу применити на објекте тог типа. 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

У Скали се промењиве дефинишу преко кључне речи *var*. Како је Скала типизиран језик, свака промењива је одређена типом. У коду 2.6 је приказан пример коришћења промењиве у Скали.

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

scala> x + 10
val res0: Int = 20

scala> x = 20
// mutated x

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

Кôд 2.6: Промењиве у Скали

Поред промењивих, у Скали постоје и именоване вредности. Дефинишу се кључном речи *val* и могу се посматрати као промењиве којима се не може променити вредност. Пример коришћења именоване вредности је приказан у коду 2.7.

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

scala> x + 20
val res0: Int = 30

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

Кôд 2.7: Именоване вредности у Скали

2.5 Контрола тока

Скала поседује уграђене стандардне наредбе за контролу тока, *if* за гранање, *while* за петље и *for* за итерирање кроз колекције [32]. У примеру 2.8 су те наредбе приказане у скалиној синтакси.

```
if (bool izraz) {
  // izraz je evaluiran u istinitu vrednost
} else {
  // izraz je evaluiran u neistinitu vrednost
}

while (bool izraz) {
  // dok se izraz evaluira u istinitu vrednost
}

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

Кôд 2.8: Наредбе за контролу 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. Синтакса дефиниције функције је приказана у коду 2.9.

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

Кôд 2.9: Дефиниција функције у Скали

У коду 2.10 је приказана функција која сабира два цела броја и враћа добијени резултат.

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

Кôд 2.10: Дефиниција функције која сабира два цела броја

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

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

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

Кôд 2.11: Краћи запис функције *saberi* и пример њеног позива

Тип повратне вредности се у неким случајевима ипак не сме изоставити [32]. На пример, када се користи рекурзија. Такође, функција не мора да враћа никакву вредност. У том случају се повратни тип означава са *Unit*.

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, је приказан у коду 2.12.

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

Кôд 2.12: Тип функције *saberi*

Експлицитно навођење типова дозвољава декларацију функција вишег реда, функција које као аргументе имају друге функције. Пример 2.13 приказује 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 2.13: Функција вишег реда

Тип првог аргумента ове функције одговара типу функције *saberi*, па се она може проследити новонаписаној функцији.

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

Кôд 2.14: Прослеђивање функције функцији

2.7 Објектно оријентисана својства језика

У овом одељку ће бити детаљније описана објектно оријентисана парадигма језика Скала.

Класе

Као и у Јави, у Скали класа представља шаблон према коме се праве објекти. Да би се креирао објекат дате класе, користи се кључна реч *new*. У коду 2.15 је приказан пример дефиниције и инстанцирања класе (напомена: усправна линија у интерпретеру за Скалу представља нови ред док

ознака @ праћена знаковима након назива класе представља инстанцу класе у меморији).

```
scala> class MyClass {
     | }

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

Кôд 2.15: Дефиниција и инстанцирање класе у Скали

Унутар класе се дефинишу поља (енг. *fields*) и методе (енг. *methods*), који се заједно једним именом називају чланови (енг. *members*) [32]. 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.

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

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

Кôд 2.16: 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***, приступ унутар класе;
- ***protected***, приступ унутар класе и класа које наслеђују ту класу;
- ***public***, приступ изван класе (подразумевана вредност која се не наводи).

Поља се могу дефинисати ван тела класе, што је и Скалин стандард (кôд 2.17). Због тога се класа може написати уз помоћ мањег броја линија.

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

Кôд 2.17: Дефиниција поља ван тела класе

У претходном примеру, поље *field* поседује подразумевану вредност која ће се том пољу увек доделити приликом инстанцирања класе [32]. Међутим, она се не мора навести и, уколико је то случај, пољима се мора експлицитно доделити вредност приликом инстанцирања. Пример је приказан у коду 2.18.

```
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 2.18: Инстанцирање класе без подразумеваних вредности поља

Наслеђивање

Наслеђивање се остварује на исти начин као у Јави, преко кључне речи *extends* [32]. Инстанцирање поља наткласе из поткласе се дефинише у самој дефиницији наслеђивања, након речи *extends* (Пример 2.19). Сва поља наткласе која немају подразумеване вредности се инстанцирају на овај начин. Предефинисање чланова наткласе се врши на исти начин као у Јави, преко кључне речи *override*.

```
// natklasa
scala> class MyClass(private val field: Int)

// potklasa
scala> class MyExtendedClass(
     |   private val fieldForParent: Int,
     |   private val newField: Int
     | ) extends MyClass(fieldForParent) // prosledjivanje
   vrednosti natklasi
```

```
class MyExtendedClass
```

Кôд 2.19: Наслеђивање у Скали

У претходном примеру ће се приликом инстанцирања поткласе инстанцирати и поља наткласе. Да би се непотребно заузимање меморије избегло, потребно је изоставити навођење речи *val* (или *var*) испред заједничког поља приликом дефинисања поткласе (кôд 2.20). У овом случају, поље које се прослеђује наткласи мора имати исти идентификатор у потклаcи и натклаcи.

```
// natklasa
scala> class MyClass(private val field: Int)

// potklasa
scala> class MyExtendedClass(
     |   field: Int, // polje za prosledjivanje
     |   private val newField: Int
     | ) extends MyClass(field) // prosledjivanje vrednosti
    natklasi
class MyExtendedClass
```

Кôд 2.20: Наслеђивање изостављењем речи *val*

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

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

Кôд 2.21: Апстрактна класа у Скали

Синглтон објекти

За разлику од Јаве, у Скали не постоје статичка поља. Уместо тога постоје синглтон објекти (енг. *singleton object*) [32]. Дефинишу се на исти начин као и

класе, с тим што се користи кључна реч *object* уместо *class*. Добили су име по томе што представљају класу која има тачно једну инстанцу. Инстанцирање објекта се извршава аутоматски. Сви чланови објекта се могу посматрати као статички чланови у Јава класи.

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

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

Кôд 2.22: Коришћење синглтон објекта

Метод *main*

Да би се апликација написана у Скали покренула потребно је дефинисати објекат који у себи садржи метод *main* [32]. Тај метод представља улазну тачку у сваку апликацију написану у Скали.

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

Кôд 2.23: Пример метода *main*

Својства

Основна јединица наслеђивања у Скали се назива својство (енг. *Trait*) [32]. Унутар својства се наводе поља и методи који се могу користити у класама које их имплементирају, односно наслеђују. Разлика између наслеђивања својства и класе је та што је дозвољено наследити једну класу, док је могуће наследити више од једног својства. Скала својство је веома слично Јавином интерфејсу, са разликом да својство може садржати и дефиниције метода и поља, а не само декларације. Међутим, својство је више од тога и унутар њега се може урадити све што се може урадити унутар Скала класе.

Дефинише се на исти начин као и класа с тим што се уместо кључне речи *class* користи реч *trait* (кôд 2.24). Унутар својства се декларишу и дефинишу поља и методи које класе које га имплементирају могу користити.

```
scala> trait MyTrait {
     |   def myMethod(): Unit
     |   val x: Int = 10
     | }
trait MyTrait
```

Кôд 2.24: Скала својство

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

```
scala> class MyClass extends MyTrait
class MyClass

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

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

Кôд 2.25: Додавање својства класи

Уколико класа којој се додељује својство већ наслеђује неку класу или неко друго својство, додељивање се мора извршити преко кључне речи *with* [32]. Свако ново својство које се додаје у овом случају се мора додати након нове речи *with*. Примери су приказани у коду 2.26.

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

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

Кôд 2.26: Наслеђивање више својстава

2.8 Колекције

Скала поседује велики број уграђених колекција, променљивих и непроменљивих. Неке од њих су низови, листе и торке.

Низови

Скала низ (енг. *array*) је променљива структура која представља низ података [32]. Променљива је у смислу да се вредности елемената у низу могу мењати, док је број елемената фиксиран. Сваки низ садржи елементе истог типа и може се креирати навођењем иницијалних елемената или његове дужине. Уколико се наведе дужина, сви елементи ће бити иницијализовани на подразумевану вредност жељеног типа.

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

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

Кôд 2.27: Инстанцирање низа у Скали

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

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

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

scala> a(0) = 100

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

Кôд 2.28: Приступ и измена елемента низа

Листе

Скала листе представљају непроменљиву колекцију елемената истог типа [32]. Разлика листе у Скали у односу на Јавину је та што је Скала листа увек непроменљива, док Јава листа може бити променљива.

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

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

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

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

Кôд 2.29: Пример листе у Скали

Торке

Непроменљива колекција која садржи елементе различитог типа се назива торка (енг. *Tuple*) [32]. Ова структура података се може користити када је потребно вратити више различитих вредности функције. Торка се инстанцира навођењем елемената између заграда. Елементу се приступа оператором _X где је X редни број елемента унутар торке.

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

scala> t._1
val res0: Int = 1

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

Кôд 2.30: Пример торке у Скали

Глава 3

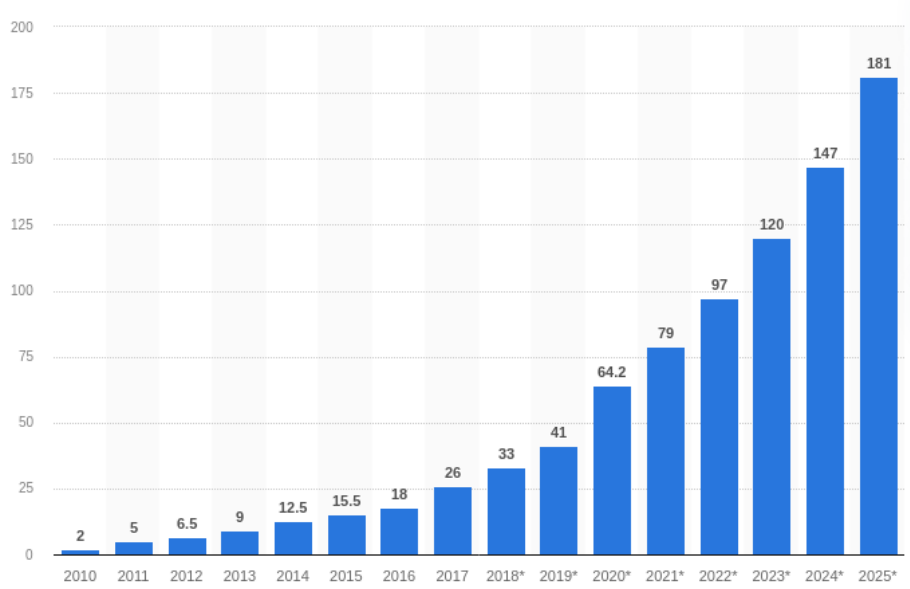
Дистрибуирана обрада података

У последњих неколико година се генерише огромна количина података [29]. Друштвене мреже, видео садржај, куповина преко интернета и Интернет ствари (енг. *Internet of Things*) на дневном нивоу производе петабајте података, а у будућности се очекује пораст тог тренда. Како је често проблем обрадити огромне количине података на једној машини, индустријски стандард су постали кластери који раде са подацима на дистрибуиран начин.

3.1 Доба података

Према истраживању [29] приказаном на једној од водећих интернет платформи за податке који се користе у пословању, *Statista*, количина података која се производи се тренутно може мерити у зетабајтима (милионима петабајта). Исто истраживање приказује да ће се наредних година тај број удвостручити. Приказ тренда пораста генерисања података је приказан на слици 3.1.

Корист од података је огромна и велики број компанија их користи за разне намене, 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. Подаци су се раније, док још увек нису генерисани у количинама у којима се то дешава данас, обрађивали на појединачним машинама, али се убрзо испоставило да такав приступ има своја ограничења.



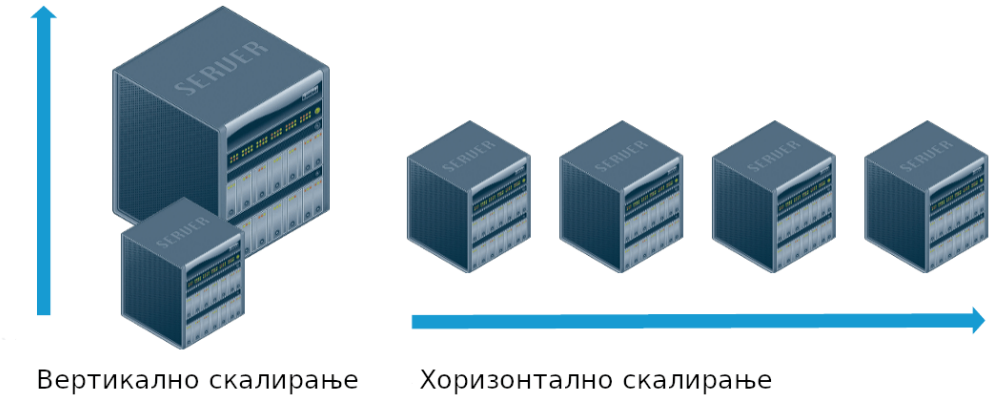
Слика 3.1: Количина података по години у зетабајтима [29]

3.2 Скалирање система

У контексту система, скалирање означава могућност система да се прилагоди количини података који се уз помоћ њега обрађују. Постоје два начина скалирања уређаја који врше обраду података (слика 3.2). Први начин је вертикално скалирање (енг. *vertical scaling*) или *scale-up* [38]. У овом приступу се унапређује једна машина, на пример, додавањем веће количине меморије или појачавањем снаге процесора. Предност овог приступа је што се након унапређења машине не мора мењати логика апликација које се на њој извршавају. Али негативна особина је што постоји ограничење до ког се машина може унапредити, па стога постоји и ограничење у количини података које она може обрадити. Такође, у случају грешке, цео систем престаје са радом, пошто се састоји од само једне машине.

Други приступ је хоризонтално скалирање (енг. *horizontal scaling*) или *scale-out* [38]. У овом случају се не унапређује једна машнина, већ се, уколико је потребна додатна снага, додаје нова машина у систем. Добра особина овог приступа је што је често јефтиније додати неколико нових машина у систем него унапредити процесор неколико пута на истој машини. Још једна веома добра одлика је ефикасност. Када постоји неколико машина могуће је на свакој од њих обрађивати један део података, што је огромна предност у

односу на вертикално скалирање. Међутим, хоризонтално скалирање доноси додатан скуп проблема. Потребно је имплементирати цео систем на потпуно другачији начин, омогућити машинама да раде заједно и координисати их, као и обрадити грешке који се могу десити на појединачним машинама. Како су наведене предности значајне, а мане се могу превазићи, данашњи стандард у обради великих количина података је хоризонтално скалирање.



Слика 3.2: Врсте скалирања система

3.3 Организација дистрибуираних система

У оквиру хоризонталног скалирања свака машина у систему обрађује један део података и на тај начин доприноси коначном резултату, због чега машине морају да комуницирају једна са другом [38]. Поред тога, могуће је да постоје подаци који су потребни свим машинама у систему, што може довести до такмичења уређаја за приступ тим подацима. Уколико се подаци налазе на само једној машини у систему, све друге машине ће јој приступити, тако да су могућности система у том случају ограничене могућностима те једне машине којој све остале приступају. Поред тога, на тој машини се може догодити некакав проблем због ког она може да престане да функционише, што би изазвало престанак рада целог система.

Да би се потенцијални проблеми избегли, систем треба 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, треба направити систем који се као целина понаша исправно, чак и при појави фаталних грешака.

У оваквим системима акценат је на софтверу, а не на хардверу и идеја је да се систем може направити од уређаја који су релативно јефтини и масовно доступни [38]. Такође, циљ је да се избегава премештање података међу уређајима, па се подаци, уколико је то могуће, обрађују на машини на којој се налазе.

3.4 Систем *Hadoop*

Први широко доступан систем који поседује претходно наведене карактеристике је развила компанија *Google* која је 2003. године објавила научни рад на ту тему [34]. У раду је представљен дистрибуирани фајл систем, назван *Google file system* или скраћено *GFS*. Систем је написан у програмском језику *C++*. Намена овог система је да се користи за складиштење великих количина података. Већ следеће године, *Google* је објавио нови научни рад о парадигми за ефикасну обраду велике количине података на кластеру [15]. Парадигма је названа *MapReduce* и њена намена је да се користи за обраду података складиштених у *GFS*-у.

Недуго након тога, уз помоћ научних радова компаније *Google*, настао је пројекат отвореног кода (енг. *open source*) назван *Hadoop* са идејом да имплементира карактеристике које поседују *Google*-ови *GFS* и *MapReduce* и да се као такав користи за складиштење и ефикасну обраду падатака на кластеру сачињеном од релативно јефтиних машина [38]. Највећи делови система *Hadoop* су *Hadoop distributed file system*, скраћено *HDFS*, и парадигма *MapReduce*, који су заправо јавно доступни еквиваленти *Google*-ових технологија. Њихови логои су приказани на слици 3.3.

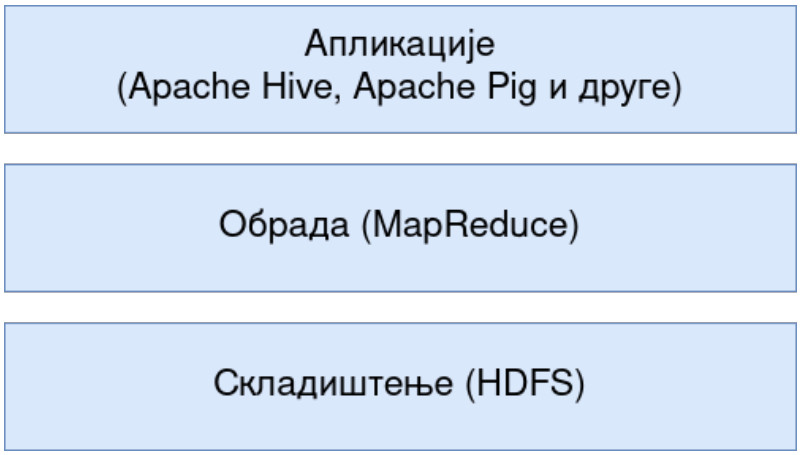
Укратко, *HDFS* је фајл систем који користи хоризонтално скалирање машина за складиштење огромних количина података [38]. Због боље поузданости користи репликацију, где се сваки фајл копира неколико пута и онда се те копије чувају на различитим уређајима у систему.

MapReduce је парадигма за обраду података, која се састоји из два дела названа *Map* и *Reduce*, по којима је и добила име [38]. Улога дела *Map* је да чита података из *HDFS*-а у деловима и трансформише их, док *Reduce* прикупља резултате обраде фазе *Map* и спаја их у један. Обрада се извршава

на истим машинама *HDFS*-а на којима се подаци и налазе, чиме се избегава њихово премештање на неку другу машину.

Слика 3.3: Логои *HDFS*-а и *MapReduce*-а

Поред поменуте две компоненте, постоји и трећа, а то су *HDFS*-апликације [39]. Оне се надовезују на *HDFS* и *MapReduce* тако што их користе за, редом, складиштење и обраду података. Најпознатије су *Apache Hive* [18] и *Apache Pig* [20], али поред њих постоје и многе друге, само мање заступљене. На слици 3.4 су приказане компоненте *Hadoop*-а.



Слика 3.4: Упрошћен приказ *Hadoop*-а

3.5 Дистрибуирани фајл систем *HDFS*

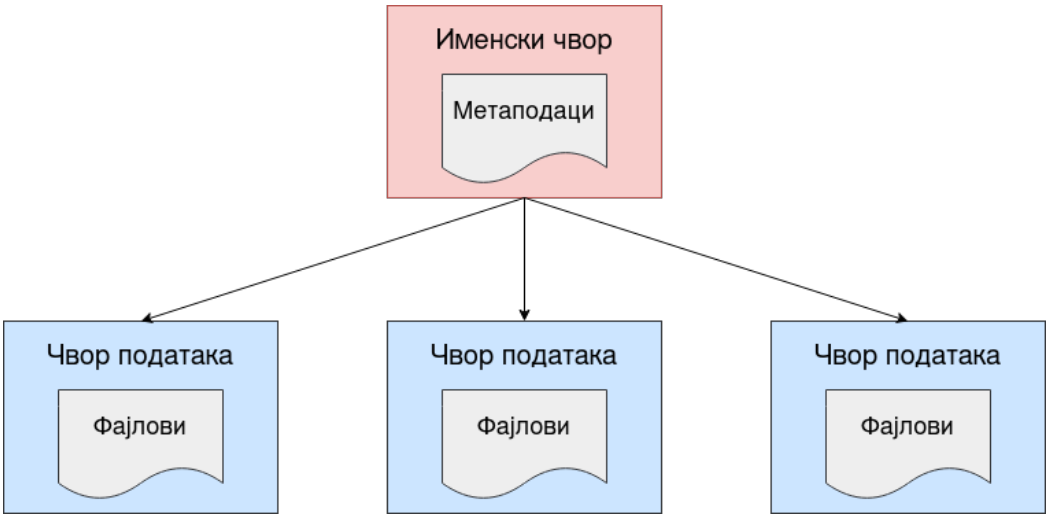
HDFS, скраћено од *Hadoop distributed file system* је дистрибуирани систем, што значи да складишти податке на више машина које ће се у даљем тексту звати чворови (енг. *nodes*).

Структура *HDFS*-а

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

Унутар система *HDFS* се налази један примарни именски чвор чија је улога да управља фајл системом и да регулише приступ подацима који се налазе на њему [24]. Он садржи информације о фајловима, као што су, између осталих, име, локација у систему где се фајл налази, последњи датум измене фајла као и правила приступа. Поред примарног, *HDFS* може имати и неколико секундарних именских чворова који представљају резервне копије.

Чворови података имају улогу да складиште фајлове система [24]. Поред тога, на овим чворовима се извршава обрада података. Чворови података су задужени за операције над фајловима као што су читање, мењање и брисање. Они ће извршити неку од тих операција само када им именски чвор то нареди.



Слика 3.5: Врсте чворова у *HDFS*-у

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

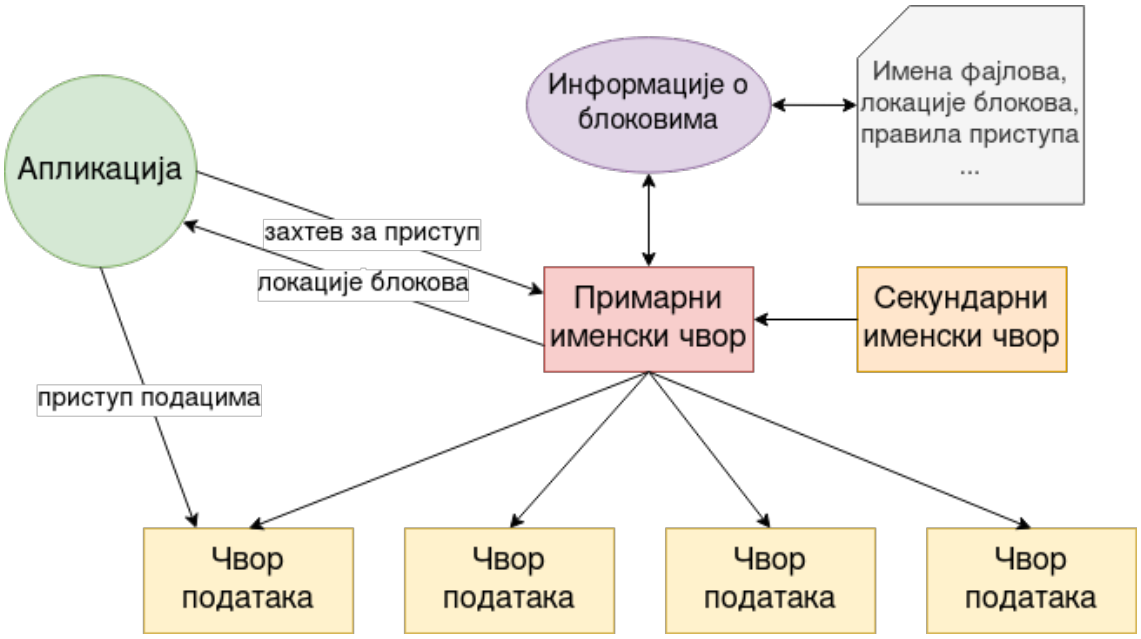
Основне карактеристике *HDFS*-а

Сваки фајл у *HDFS*-у је подељен на делове који се називају блокови чија је величина обично 128 мегабајта [24]. Блокови се често не налазе на истим чворовима у систему, што значи да се један фајл чува на неколико физички раздвојених машина. Ту може да настане проблем због тога што једна од тих машина може да се поквари и због тога престане са радом. У том случају ће сви блокови складиштени на тој машини нестати. Да би се губљење фајлова избегло, *HDFS* сваки блок реплицира неколико пута и након тога оригинални блок и његове реплике распоређује по систему. Ако један од блокова фајла неочекивано нестане, увек је могуће приступити једној од његових реплика. Генерисане реплике се чувају на чворовима података, док се информације о томе ком фајлу реплике припадају налазе на именском чвору.

Блок ће се увек реплицирати одређен, фиксиран, број пута. Чворови података повремено шаљу сигнале именском чвору о доступности реплика. На тај начин ће именски чвор увек имати информацију о томе колико је пута сваки блок реплициран у систему и на основу тога може да, уколико тај број падне испод неке задовољавајуће вредности, направи нове реплике тог блока [24].

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

HDFS је систем за кога важи *пиши једном, читај више пута* (енг. *write-once, read-many).* Када се фајл постави унутар *HDFS*-а више се не може мењати [38]. Уколико се фајл мора изменити долази до креирања новог фајла који замењује стари. Иако такав приступ није ефикасан, апликације које обрађују велике количине података се обично заснивају на томе да се подаци не мењају, па се очекује да за променама неће бити потребе или ће такви случајеви бити ретки. Такође, још једна од особина *HDFS*-а је да има добре перформансе у случајевима када је потребан велики проток података,



Слика 3.6: Основне *HDFS*-компоненте

на пример у случају читања великих фајлова.

3.6 Парадигма *MapReduce*

MapRreduce је парадигма која се користи за обраду података који су складиштени у *HDFS*-у [38]. Користи приступ подели и завладај (енг. *divide and conquer*) приликом обраде тако да више машина паралелно обрађује по један део података.

Парадигма је заснована на концептима функционалног програмирања и функцијама које се често користе у обради низова и листи. Те фунције су *map* и *reduce*. Прва од постојеће листе креира нову тако што на сваки елемент листе примени неку фунцију и од њега направи нови елемент. Друга од целе листе производи једну вредност. На истим принципима фунционише и *MapReduce*.

MapReduce обрађује податке у неколико фаза [39]. Прво, подаци се читају из *HDFS*-а и након тога прослеђују машинама које се зову мапери (енг. *mappers*). Те машине паралелно производе скуп привремених података који се након тога распоређују, сортирају и шаљу машинама које се зову редуктори (енг. *reducers*). Фаза која распоређује податке се назива фаза мешања

и сортирања (енг. *shuffle and sort*). Задатак редуктора је да приме подскуп података и да паралелно произведу једну вредност од истих. На самом крају се резултат свих редуктора комбинује и добија се резултат читавог процеса *MapReduce*, другачије званог и *MapReduce*-задатак (енг. *task*). Могуће је, уланчавањем, комбиновати *MapReduce*-задатке, тако да излаз из једног буде улаз у други. Скуп повезаних *MapReduce*-задатака се назива *MapReduce*-апликација.

MapReduce из аспекта функција

Из аспекта функција, фазе пресликавања (енг. *map*) и редуковања (енг. *reduce*) се могу посматрати на следећи начин. Подразумевани формат је *(кључ, вредност)* за који ће се због једноставности користити ознака *(k, v)*. Током фазе пресликавања подаци се читају из *HDFS*-а и деле на делове на које се паралелно примењује функција *map* дефинисана од стране програмера. Паралелизам се постиже тако што се сваки део обрађује на засебној машини, маперу. Фаза пресликавања као улаз прима парове *(k, v)* и производи листу истог формата [39].

$$list(k_1, v_1) \rightarrow map(list(k_1, v_1)) \rightarrow list(k_2, v_2)$$

Након тога се листе генерисане од мапера групишу тако што се за сваки кључ прави једна група коју ће обрадити један редуктор. Овај део обраде је мешање и сортирање.

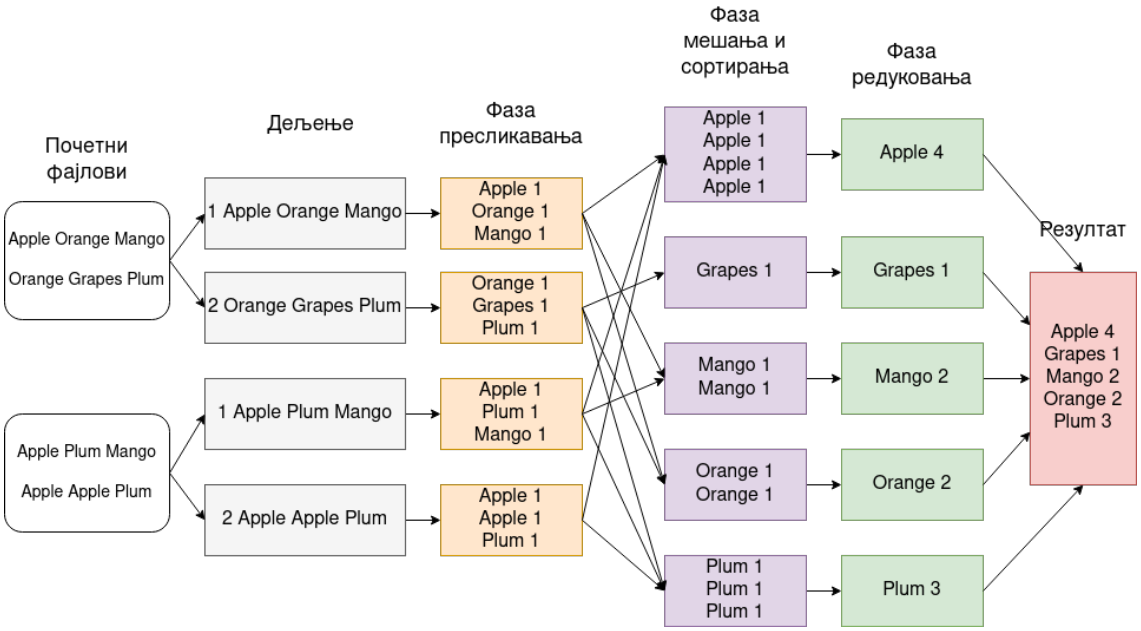
$$list(k_2, v_2) \rightarrow shuffleAndSort(list(k_2, v_2)) \rightarrow k_2, list(v_2)$$

У последњој фази се на сваку од креираних група примењује функција *reduce* која производи једну вредност за сваку групу [39]. Овај процес је паралелизован и није могуће две групе података са различитим кључевима обрађивати на истој машини у истом тренутку. Паралелна редукција је могућа само ако је редукција дефинисана као асоцијативна и комутативна операција. Фаза редуковања прима кључ и листу вредности које му одговарају и као резултат производи једну вредност формата *(k, v)*.

$$k_2, list(v_2) \rightarrow reduce(k_2, list(v_2)) \rightarrow (k_3, v_3)$$

Коначан резултат се добија комбиновањем резултата свих редуктора и може се уписати у *HDFS* или се искористити као улаз у други *MapReduce*-задатак. У *MapReduce*-апликацијама задатак програмера је да опише како ће се извршавати фазе пресликавања и редуковања, док ће се систем *Hadoop* побринути за све остало: читање података, сортирање, паралелизацију, координацију и извршавање послова [38].

Пример *MapReduce*-апликације је приказан на слици 3.7 где је представљен процес пребројавања броја појављивања сваке речи у тексту. У приказаном примеру је улаз у мапере форматиран тако да је кључ редни број линије фајла, док је вредност текст линије. Улога мапера је да поделе текст на речи и да од њих направе листе парова формата *(реч, 1)*. Након тога се парови који имају исту реч премештају на засебне редукторе који израчунавају колико пута се у почетном скупу појављује свака реч, сумирањем јединица.

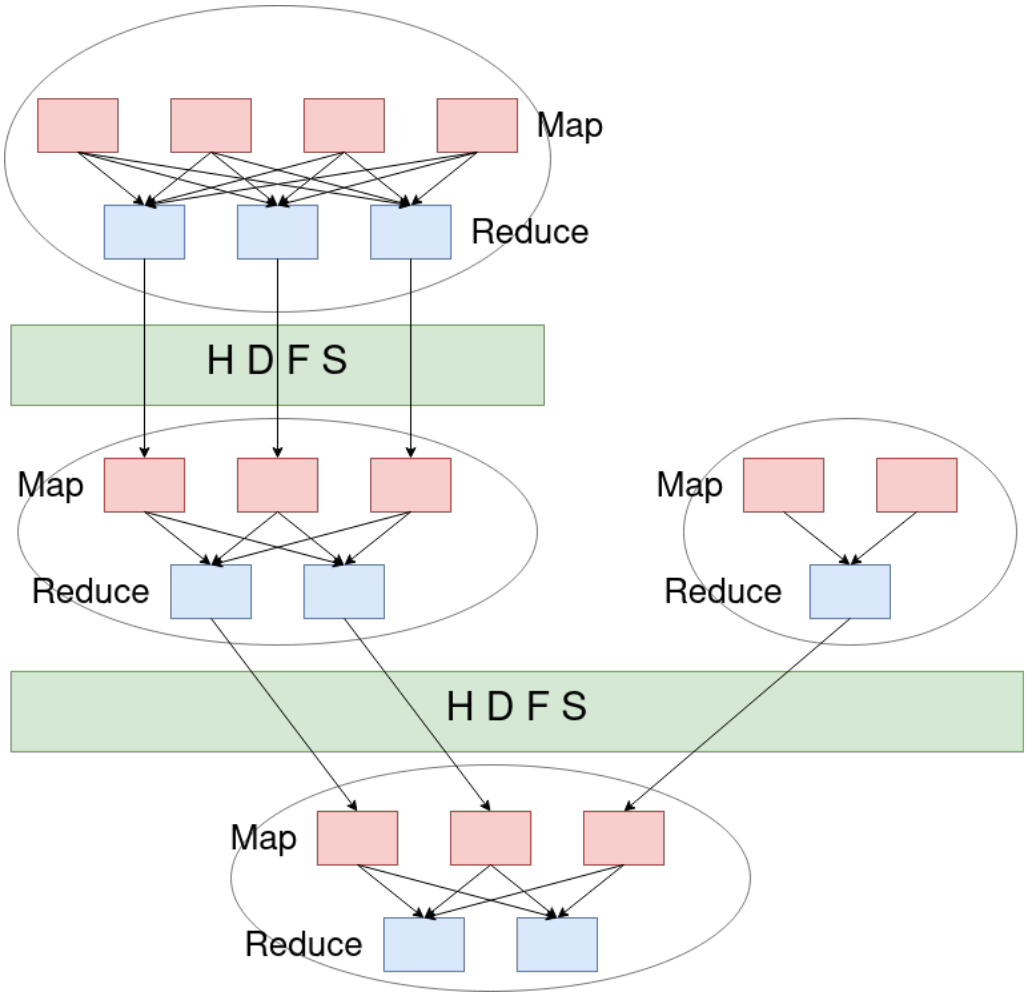


Слика 3.7: Пример *MapReduce*-апликације

Недостаци парадигме *MapReduce*

MapReduce-апликација се састоји од ланца *MapReduce*-задатака, таквих да излаз једног задатка представља улаз у други (слика 3.8). Међутим, такав приступ има цену, а то је да се излаз генерисан од стране једног *MapReduce*-задатка чува унутар *HDFS*-а, одакле му приступају други *MapReduce*-задаци

којима је тај излаз потребан [39]. Другим речима, међурезултати задатака се чувају на диску, што ствара додатне улазно/излазне операције и тиме успорава извршавање целокупне апликације. Поред тога, унутар парадигме *MapReduce* не постоји аутоматски начин да се задаци заједно оптимизују, на пример комбиновањем, већ је за то задужен програмер.



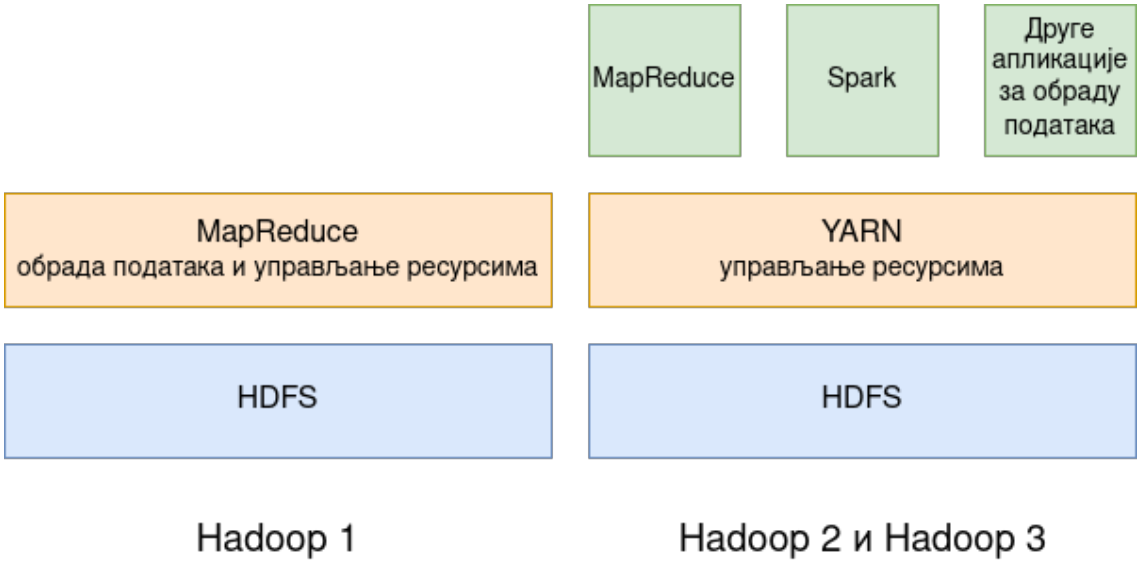
Слика 3.8: Пример ланца *MapReduce*-задатака

Због поменутих недостатака се парадигма *MapReduce* у данашње време ретко користи. Потиснута је од стране других технологија и алата, међу којима је и *Apache Spark* (поглавље 4).

3.7 Преговарач ресурса *Apache Yarn*

У првој верзији *Hadoop*-а, *MapReduce* је поред обраде великих количина података, за шта је примарно и намењен, имао додатне задатке, а то су заказивање *MapReduce*-задатака и алокација и управљање ресурсима који су *MapReduce*-апликацији потребни [39]. Таква архитектура је знатно отежавала конструкцију апликација које користе *MapReduce*, па су због тога, у другој верзији *Hadoop*-а, одговорности *MapReduce*-а раздвојене. *MapReduce* је постао алат искључиво за обраду података, док је управљање ресурсима предато новој апликацији, са идејом да је *MapReduce* током извршавања користи.

Резултат је менаџер ресурса (енг. *resource manager*) отвореног кода назван *Yarn* [22] или *yet another resource negotiator*. Његова улога је да распоређује задатке апликација које користе *Hadoop*, али и да управља ресурсима који су тим апликацијама потребни [39]. Конструисан је да не буде специфичан само за *MapReduce*, већ пружа интерфејс ка *Hadoop*-у разним апликацијама међу којима је и *Apache Spark*. Разлика у архитектури у различитим верзијама *Hadoop*-а је приказана на слици 3.9.



Слика 3.9: Разлика у архитектури између *Hadoop* верзија

Архитектура *Yarn*-а

Улога *Yarn*-а је искључиво да распореди извршавање задатака на кластеру и обезбеди им ресурсе потребне за њихово извршавање [39]. Све остало, попут надгледања система, праћења прогреса апликација, обраде грешака и сличног, је имплементирано у коду апликације која га користи.

Састоји се од две главне компоненте, менаџера ресурса (енг. *resource manager*) и менаџера чвора (енг. *node manager*) [39]. Улога првог менаџера је да управља ресурсима читавог кластера, док други управља ресурсима машине на којој је покренут. То значи да ће кластер имати један менаџер ресурса и више менаџера чвора, по један за сваку машину у кластеру. Заједно, они управљају контејнерима (енг. *container*), апстракцијом меморије, процесорске снаге и улазно-излазних операција потребних да би се извршио један део апликације на кластеру.

Менаџер ресурса је најбитнија компонента *Yarn*-а и одговоран је за извршавање сваке апликације на кластеру [39]. Састоји се од две компоненте, заказивача (енг. *scheduler*) и менаџера апликације (енг. *application manager*). Прва регулише распоред извршавања апликација, док друга прихвата апликације и преговара о алокацији првог контејнера који им је потребан.

Апликација која се покреће преко *Yarn*-а се састоји из два дела. Први део је кôд који треба извршити на кластеру, док се други зове власник апликације (енг. *application master*) [39]. Његова улога је да преговара о ресурсима и прати прогрес и статус апликације. *Yarn* нема информацију на који начин је успостављена комуникација између мастера апликације и кода који се извршава. Приказ архитектуре и компоненти *Yarn*-а у случају извршавања две апликације на кластеру је приказан на слици 3.10 (напомена: извршавају се две апликације, што значи да постоје два власника апликације).

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

1. Клијент пријављује апликацију.
2. Менаџер ресурса алоцира контејнер на чвору у коме се покреће власник апликације.
3. Власник апликације се региструје код менаџера ресурса.

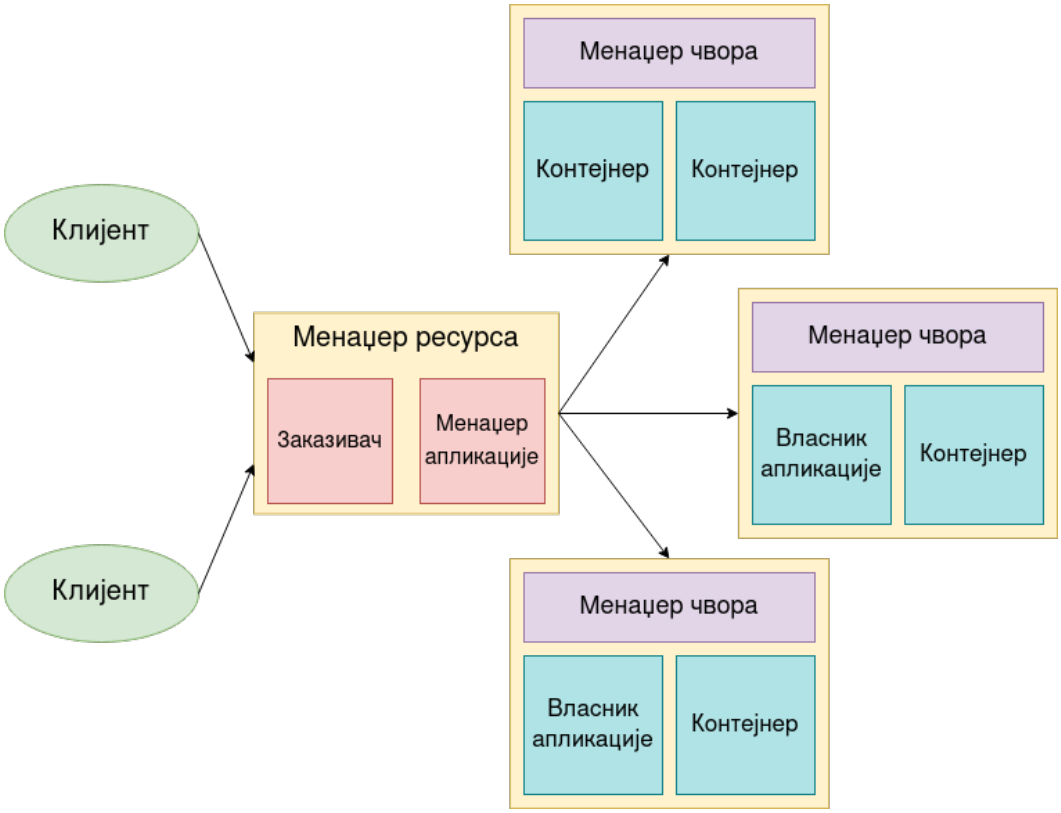
4. Власник апликације преговара о контејнерима са менаџером ресурса. У исто време, заказивач распоређује извршавање делова апликације.

5. Власник апликације комуницира са менаџером чвора о покретању потребних контејнера за извршавање апликације.

6. Кôд апликације се извршава унутар контејнера.

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

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



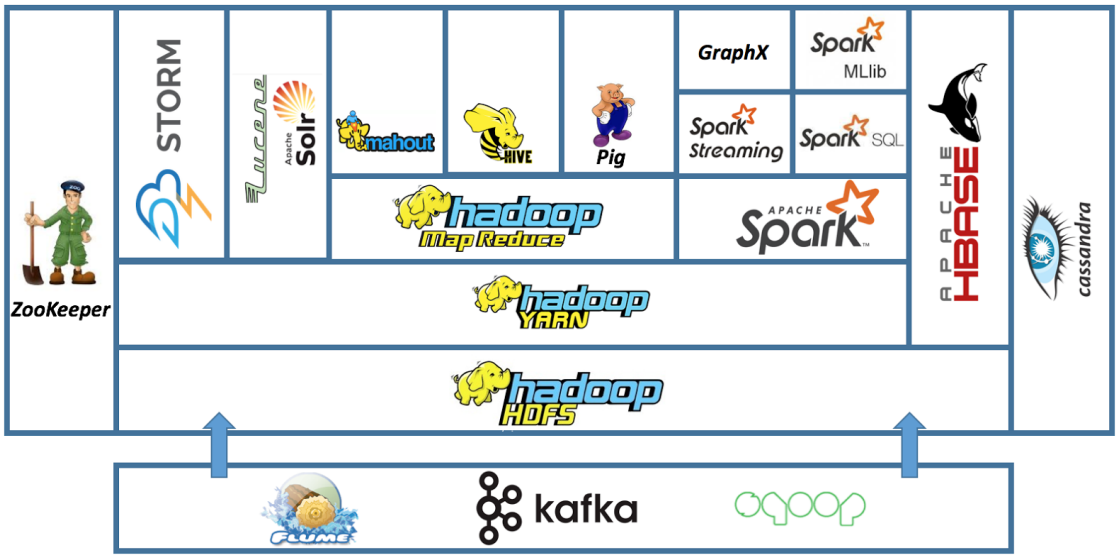
Слика 3.10: Компоненте *Yarn*-а у случају извршавања две апликације

Yarn има одговорност да омогући правилно извршавање апликацијама које се извршавају на *HDFS*-у па стога мора обрадити грешке које се могу појавити [39]. На пример, могуће је да једна од машина у кластеру престане се радом и тако постане неупотребљива. Када се то деси, менаџер ресурса ће

менаџер чвора на тој машини означити мртвим и неће га више разматрати. Исто ће се десити и са контејнерима те машине. Такође, сваки контејнер који почне да користи више ресурса од оних који су му омогућени ће бити уништен, да не би изазивао проблеме другим апликацијама у систему.

3.8 Остале компоненте *Hadoop*-а

Екосистем *Hadoop* чини велики број апликација разних примена које на неки начин користе *HDFS*. Поред самог *HDFS*-а, *MapReduce*-а и *Apache Yarn*-а у њега спадају и *Apache Kafka* [19], апликација за рад са токовима података, *Apache Pig* [20] и *Apache Hive* [18], које се користе за обраду података и имплементиране су коришћењем *MapReduce*-а. Поред њих постоје, на пример, *Presto* [27], *Apache Flume* [17], *Apache Zookeeper* [23] али и многе друге. Приказ малог дела екосистема *Hadoop* је приказан на слици 3.11.



Слика 3.11: Део екосистема *Hadoop*

Глава 4

Алат *Apache Spark*

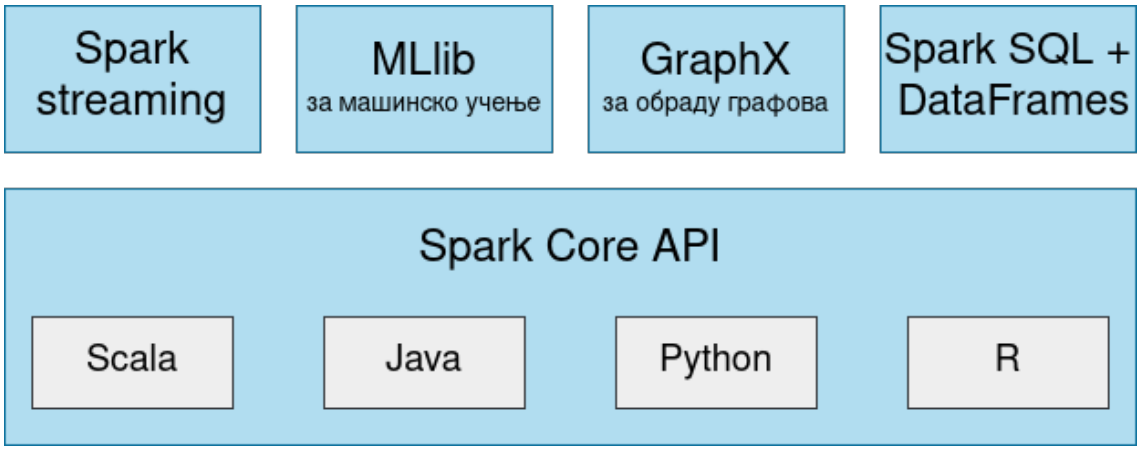
Apache Spark је алат отвореног кода. Настао је 2009. године на универзитету Беркли (енг. *Berkeley*) у Калифорнији. Написан је у програмском језику Скала и дизајниран је са идејом да користи концепте функционалног програмирања. Постао је део фондације *Apache* 2013. године. Од тада су избачене три верзије,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

Намењен је за дистрибуирану обраду велике количине података али се поред тога користи и за рад са токовима података (енг. *streaming*), машинско учење и рад са графовима [14]. Може се користити у програмским језицима Скала, Јава, *Python* и *R*. Иако је намењен за рад на кластерима, може се користити и на једној машини. На слици 4.1 су приказане компоненте *Apache Spark*-а. У овом поглављу ће детаљније бити приказане оне које се користе за обраду података.

4.1 Архитектура

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

Свака *Spark*-апликација се састоји из једног контролног процеса (енг. *driver process*) и једног или више процеса извршилац (енг. *executor process*). Контролни процес је срце *Spark*-апликације и има три задужења:



Слика 4.1: Компоненте *Apache Spark*-а

- прикупљање информација о апликацији која се извршава;
- конвертовање кода апликације у послове које треба извршити на извршиоцима;
- анализирање, распоређивање и планирање тих послова.

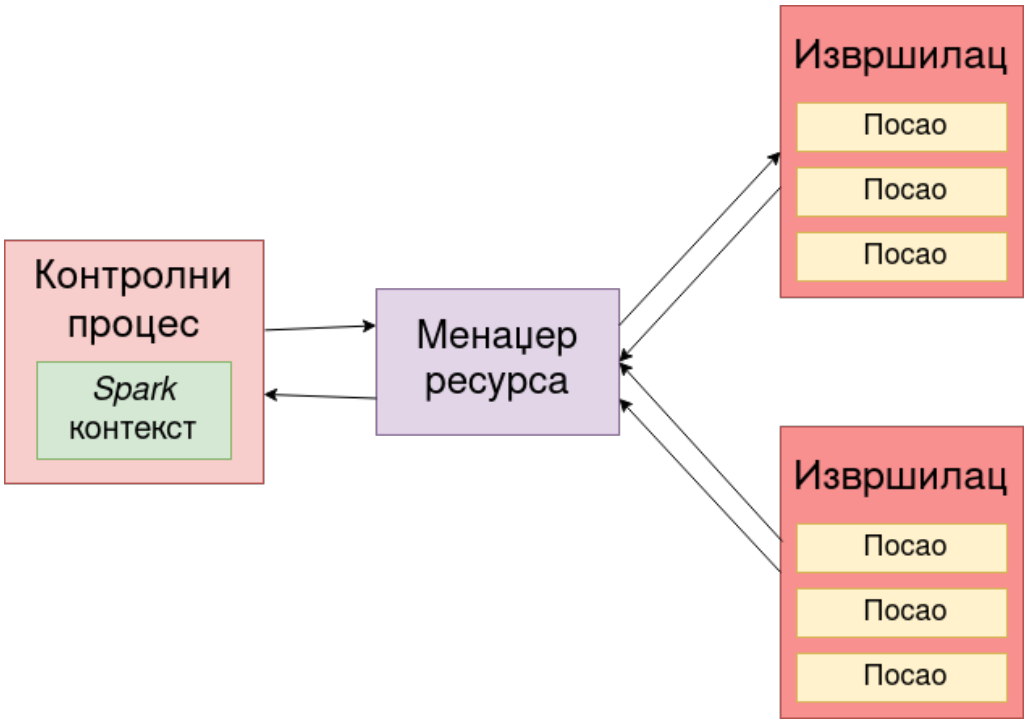
Извршилац има улогу да извршава посао који му контролни процес задаје. Поред тога, задужен је и за пријављивање стања извршавања тог посла контролном процесу.

Унутар контролног процеса постоји одвојен процес назван *Spark* контекст (енг. *Spark context*) [14]. Његова улога је да дефинише конекцију ка кластеру. Такође се користи и за креирање апстракција *Spark*-а названих *RDD* (поглавље 4.3). Једноставан приказ архитектуре *Spark*-а је дат на слици 4.2.

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

4.2 Партиције

Да би извршиоци могли паралелно да извршавају операције над подацима, *Spark* податке дели на делове који се називају партиције (енг. *partitions*) [14]. Партиција је део колекције података који се налази на једној машини кластера. За партиције важи да се једна партиција увек обрађује од стране једног



Слика 4.2: Архитектура *Apache Spark*-а

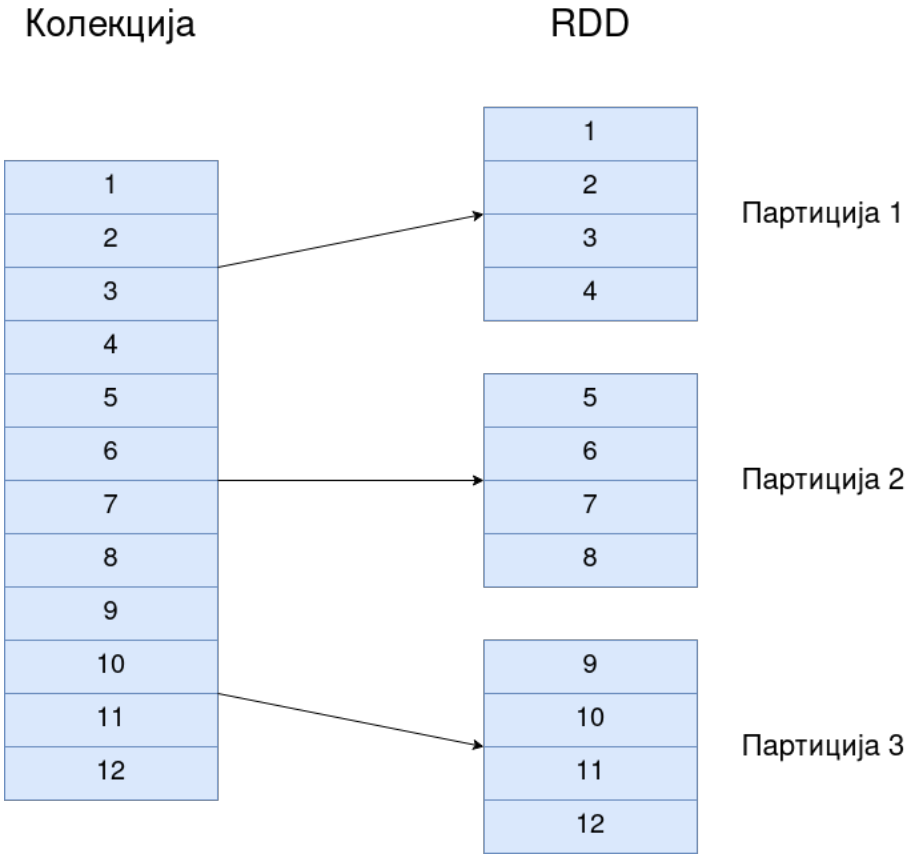
извршиоца, као и да један извршилац, у једном тренутку, обрађује податке тачно једне партиције. Дељење података на партиције у *Spark*-у је аналогно дељењу података на делове приликом извршавања фазе *map MapReduce*-а.

Уколико су подаци партиционисани само једном партицијом, биће обрађени од стране једног извршиоца у кластеру, независно од тога колико извршилаца постоји. Слично, уколико је креирано више партиција, али постоји само један извршилац, паралелизам неће постојати, због тога што постоји само једна машина која може обрадити податке.

4.3 Апстракција података *RDD*

Основна јединица рада у *Spark*-у се назива еластичан дистрибуирани скуп података (енг. *resilient distributed dataset*), скраћено *RDD*, и све операције са подацима се извршавају преко ње. *RDD* је колекција елемената за које важи да су партиционисани по машинама кластера и да се над њима паралелно могу извршавати операције [35]. Постоји неколико начина преко којих се *RDD* може креирати:

- читањем фајла који се налази на фајл систему (обично *HDFS*);
- паралелизацијом — процесом дељења у партиције колекције података програмског језика у коме се *Spark* користи (слика 4.3);
- од већ постојећег *RDD*-ја применом *Spark*-трансформације;
- кеширањем постојећег *RDD*-ја.



Слика 4.3: Креирање *RDD*-ја од колекције података

Данас се апстракција *RDD* сматра застарелом и не користи се директно, већ постоје друге које су конструисане над њом и које су је потиснуле, углавном због бољих перформанси, попут *Spark DataFrame*-а (поглавље 4.4). Битно је нагласити да се апстракција *DataFrame* заснива на истим концептима као и *RDD*, као и да се сваки *DataFrame* кôд преводи у *RDD* пре извршавања.

Трансформације

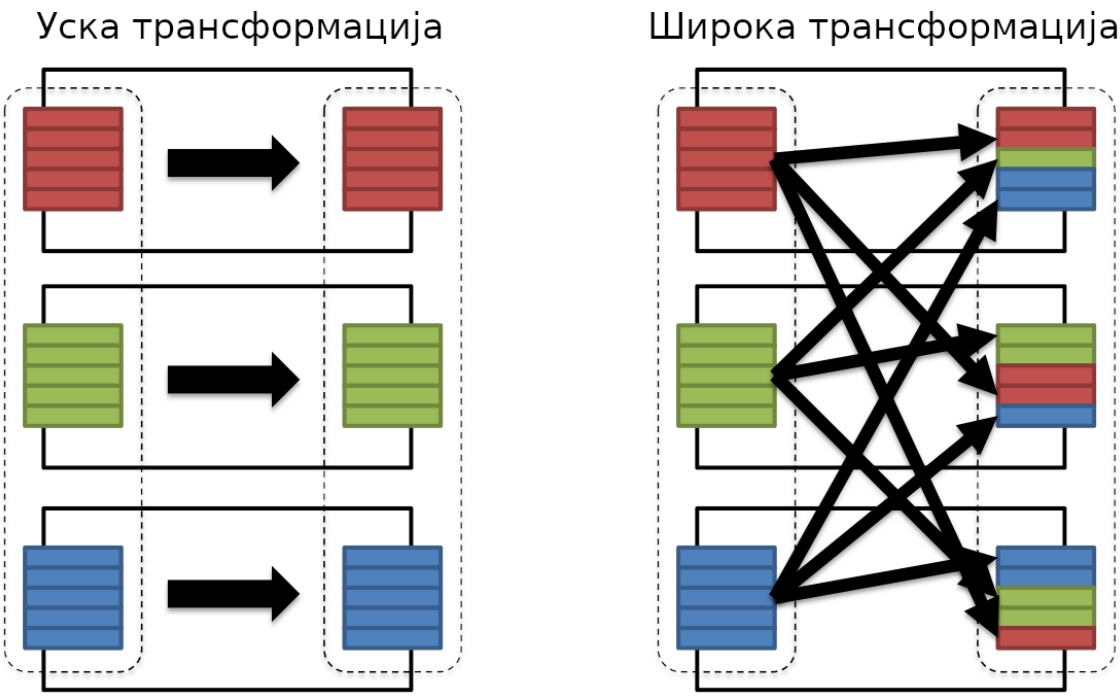
Spark је конструисан по принципима функционалног програмирања, па су све његове структуре података непроменљиве, што значи да се након креирања оне не могу мењати [14]. Пошто се подаци не могу мењати, свака операција која треба да их измени заправо креира потпуно нову структуру података. На пример, уколико постоји *RDD* којем се мењају подаци које садржи, они се неће изменити непосредно, већ ће се од постојећег *RDD*-ја направити нови који у себи садржи измењене податке.

Тај процес, где се од једног *RDD*-ја применом наредби добија други, се назива трансформација (енг. *transformation*) [14]. Пратећи функционалне концепте, трансформације немају бочне ефекте, што значи да се од једног *RDD*-ја применом истих трансформација, као резултат увек добија одговарајући *RDD*, независно од тога када се те трансформације примењују. *RDD* који трансформацијом настаје од другог *RDD*-ја се назива зависни *RDD* (енг. *dependency*).

Постоје две различите врсте трансформација, уске (енг. *narrow*) и широке (енг. *wide*) [14]. За уске трансформације важи да једна партиција у почетном *RDD*-ју доприноси настајању највише једне партиције у зависном *RDD*-ју. Са друге стране, широке трансформације су такве где једна партиција почетног *RDD*-ја учествује у конструисању више партиција зависног *RDD*-ја. Обе врсте трансформација су приказане на слици 4.4. Из приказане слике се за широку трансформацију може приметити да се подаци унутар једне партиције изворног *RDD*-ја премештају у сваку партицију зависног *RDD*-ја, слично као у оквиру фазе мешања и сортирања *MapReduce*-а. Та појава се другачије назива мешање (енг. *shuffle*).

Постоји значајна разлика у перформансама узмеђу уских и широких трансформација [14]. Код уских, *Spark* извршава операције у меморији, док код широких пише резултате на диск и поново их распоређује по партицијама, што значајно успорава извршавање.

Све трансформације у *Spark*-у припадају лењој евалуацији што значи да се не извршавају док се њихова вредности не затражи [14]. За сваки ланац трансформација, *Spark* креира *план трансформација* који се извршава тек када је потребан њихов резултат. Евалуација скупа трансформација се у *Spark*-у назива *акција* (одељак 4.3).



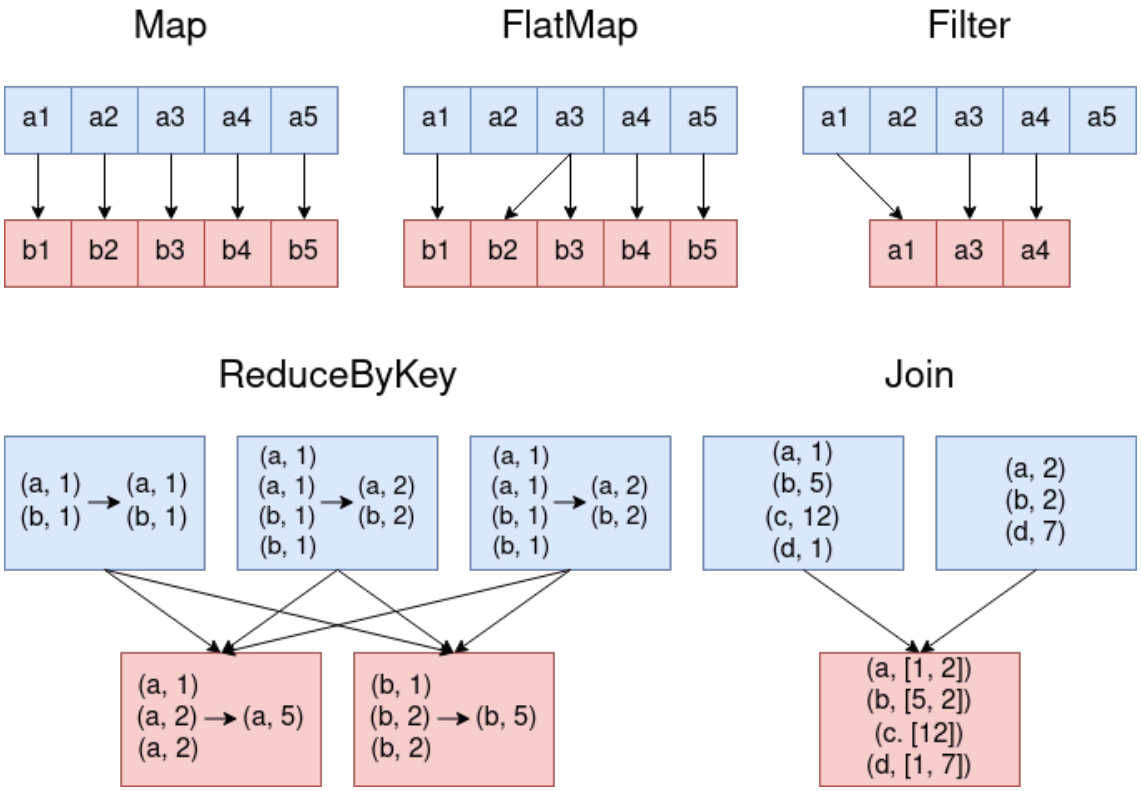
Слика 4.4: Приказ врста *Spark*-трансформација

Примери *RDD*-трансформација

У *Spark*-у постоји велики број трансформација, уских и широких. Неке од најпознатијих су приказане на слици 4.5 и то су:

- ***map***, за сваки елемент почетног скупа података производи нови, применом неке операције;
- ***flatMap***, функционише исто као *map* са тим што сваки елемент почетног скупа производи нула, један или више елемената новог скупа (уколико за сваки елемент произведе тачно један нови елемент, ова трансформација је идентична *map* трансформацији);
- ***filter***, од постојећег скупа елемената производи нови у коме се налазе они елементи почетног скупа који задовољавају некакав услов;
- ***reduceByKey***, редукује вредности са заједничким кључем — редуковање се иницијално извршава по партицији, након чега се подаци распоређују по новим партицијама и поново редукују по кључу;

- ***join***, спаја два скупа елемената у један, где ће резултат бити скуп података у коме је вредност сваког кључа унија вредности тог кључа у засебним скуповима који учествују у спајању.



Слика 4.5: Примери *RDD*-трансформација

Све *RDD*-трансформације су део *Spark Core*-а [36]. Поред поменутих, постоје и *aggregate*, *union*, *intersect*, *mapValues*, *sortByKey* али и многе друге.

Акције

Spark-акције се користе када је потребно евалуирати резултат ланца трансформација [14]. Уколико је резултат акције нека вредност, она се прослеђује контролном процесу. Постоје три врсте акција:

- акције које приказују резултат у конзоли;
- акције које исписују резултат на излаз, на пример у фајл;

- акције које пребацују податке у колекцију програмског језика у коме се користи *Spark*.

Као и трансформације, *RDD*-акције су део *Spark Core*-а [36]. Неке од најкоришћенијих акција су:

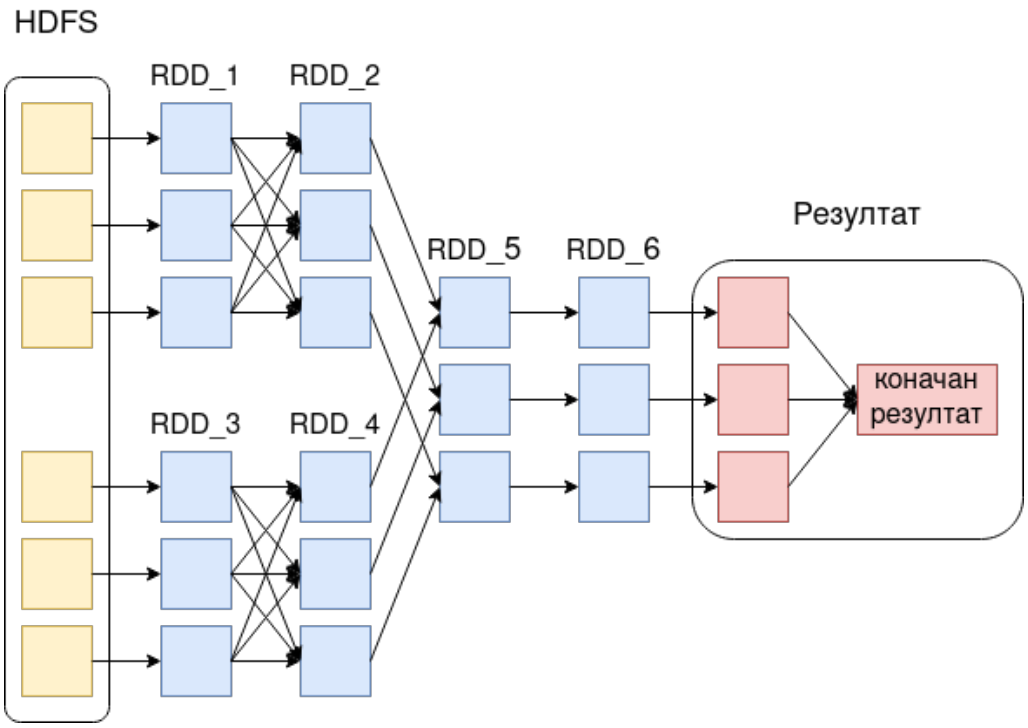
- ***count***, исписује број елемената у структури података;
- ***saveAsTextFile***, чува податке унутар *RDD*-ја у текстуални фајл;
- ***collect***, пребацује све податке *RDD*-ја у колекцију програмског језика;
- ***take***, пребацује првих N података *RDD*-ја у колекцију програмског језика.

Руковање грешкама

У току извршавања *Spark*-трансформација могућа је појава грешака које могу да резултују губитком партиција. У том случају је *Spark* у могућности да их поврати помоћу механизма који се назива граф наслеђивања (енг. *lineage graph*) у коме се чувају информације о томе од ког *RDD*-ја је сваки *RDD* у ланцу трансформација настао и применом којих трансформација. Пример једног ланца *Spark*-трансформација који у исто време представља и граф наслеђивања је приказан на слици 4.6. Из примера се може закључити да је *RDD_5* настао спајањем *RDD_2* и *RDD_4*.

Уз помоћ графа наслеђивања *Spark* може да закључи од које партиције је настала свака партиција у ланцу и уколико нека од њих нестане, може је поново направити [39]. У случају да нека партиција ланца није исправна, *Spark* ће проверити све партиције од којих је она настала. Уколико оне постоје, поново ће направити неисправну партицију од њих, примењујући потребне трансформације. У супротном ће рекурзивно прегледати изворне партиције тих партиција и тај процес ће понављати све док се не пронађе исправна партиција или се не дође до партиције која је настала директним читањем са диска. У том случају ће је *Spark* поново прочитати и након тога покренути ланац трансформација из почетка.

Процес поновне конструкције партиција је поуздан из два разлога. Први је што трансформације немају бочне ефекте, па ће се поновним креирањем увек добити одговарајући *RDD*. Други је што се изворни подаци чувају у



Слика 4.6: Пример једног *Spark*-извршавања

HDFS-у, који је поуздан, па ће се у случају поновног читања из меморије и поновним креирањем читавог ланца, увек прочитати почетна, непромењена, вредност са диска.

Кеширање

Веома битна карактеристика *Spark*-а је могућност чувања података у меморији, односно кеширање [35]. Када се *RDD* кешира, свака машина у кластеру ће у својој меморији сачувати партиције које се на њој налазе и касније их користити у акцијама или трансформацијама у којима је тај *RDD* потребан, без извршавања целог ланца трансформација из почетка. Овакав приступ знатно побољшава перформансе *Spark*-апликације. Чување у меморији се извршава тек након што *RDD* учествује у некој акцији. Кеширање је отпорно на грешке, и за поновно креирање несталих партиција кешираног *RDD*-ја се користи граф наслеђивања.

RDD се може кеширати коришћењем функција *cache* и *persist* [35]. Оне омогућавају различите нивое кеширања у зависности од тога у којој врсти меморије се партиције чувају:

- *cache* кешира податке у меморији;
- *persist* са аргументом *MEMORY_ ONLY* кешира податке у меморији;
- *persist* са аргументом *DISC_ ONLY* кешира податке на диску (овај приступ се не саветује зато што је често брже поново извршити цео ланац трансформација из почетка, него учитати кеширан *RDD* са диска);
- *persist* са аргументом *MEMORY_ ONLY_ 2* кешира податке у меморији али поред тога извршава репликацију *RDD*-ја на још једну машину кластера;
- *persist* са аргументом *DISC_ ONLY_ 2* кешира податке на диску али поред тога извршава репликацију *RDD*-ја на још једну машину кластера;
- *persist* са аргументом *MEMORY_ AND_ DISC* кешира податке у меморији уколико постоји довољно простора, а у супротном кеширање извршава на диску.

4.4 Апстракција података *DataFrame*

DataFrame је дистрибуирана колекција налик табели, са дефинисаним редовима и колонама [14]. Свака колона мора имати исти број редова и сваки ред мора имати исти број колона. Поред тога, свакој колони је додељен један тип ког морају бити све вредности које се у њој налазе.

Сваки *Spark DataFrame* садржи метаподатке који описују имена колона и њихове типове [14]. Ти метаподаци се називају шема (енг. *schema*). Шема се може дефинисати експлицитно али се може и аутоматски закључити из података који се налазе унутар *DataFrame*-а. Поред типова, у шеми се налази информација о томе да ли колона може поседовати вредности *null*. На слици 4.7 је приказан једноставан пример *DataFrame*-а и његове шеме.

У *Spark*-у постоји велики број типова који се могу доделити колонама *DataFrame*-а [14]. Постоје једноставни типови попут целих бројева, децималних бројева и ниски али постоје и сложени, попут низова, мапа и датума. Сви *Spark*-типови се могу пресликати у одговарајуће типове програмских језика у којима се он користи.

DataFrame

Име	Број индекса	Смер
Милица	1100	Математика
Петар	1101	Информатика

Шема

Име: string (nullable = false)
Број индекса: integer (nullable = false)
Смер: string (nullable = false)

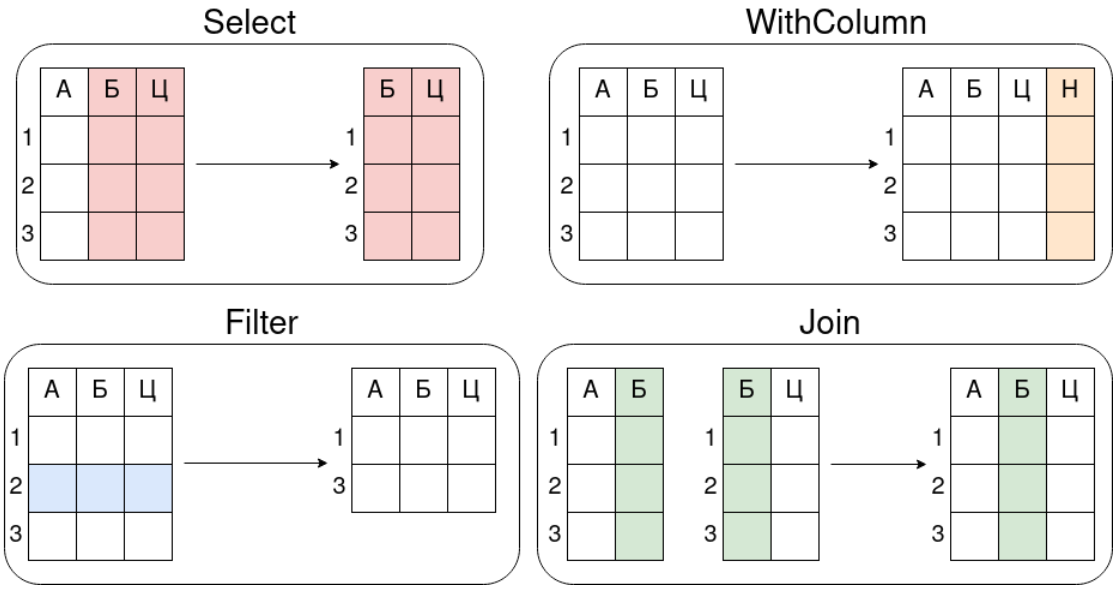
Слика 4.7: *Spark DataFrame* и његова шема

Трансформације и акције *DataFrame*-а

Све особине трансформација и акција које важе за апстракцију *RDD*, важе и за трансформације и акције *DataFrame*-а [14]. Дакле, трансформације немају бочне ефекте и лењо се евалуирају, тек када се позове акција. Такође, резултат трансформације примењене на *DataFrame* ће увек бити нови *DataFrame*. Једина разлика је у томе што *RDD* и *DataFrame* другачије представљају податке, па су им трансформације и акције другачије. Како је *DataFrame* сличан табели у релационим базама, поседује неколико трансформација које су аналогне наредбама у програмском језику *SQL*. Неке од најкоришћенијих *DataFrame*-трансформација су приказане на слици 4.8 и то су:

- ***select***, конструише нови *DataFrame* са подскупом колона почетног;
- ***filter***, конструише нови *DataFrame* са редовима почетног који задовољавају задати услов;
- ***withColumn***, конструише нови *DataFrame* додавањем колоне на почетни – вредности нове колоне се генеришу функцијом која је прослеђена као аргумент овој трансформацији;
- ***join***, спаја два *DataFrame*-а у један на основу заједничких вредности колона.

Примери акција *DataFrame*-а су *collect*, која трансформише *DataFrame* у структуру података програмског језика у коме се *Spark* користи и *show*, која



Слика 4.8: Примери *DataFrame*-трансформација

се користи за испис *DataFrame*-а на стандардни излаз [14]. Поред њих постоје и многе друге.

Кориснички дефинисане функције

Одређене трансформације као аргумент примају функцију коју користе да би добиле резултате. Једна таква трансформација је *withColumn* која генерише нову колону у односу на повратну вредност функције која јој се прослеђује. *Spark* садржи велики број уграђених функција које се могу користити и оне се налазе унутар *org.apache.spark.sql.functions* модула [8]. Уколико је потребно користити функционалност коју *Spark* нема уграђену, може се користити кориснички дефинисина функција (енг. *user defined function*), скраћено *udf*.

Udf је функција написана у Скали која је регистрована од стране *Spark*-а наредбом *udf*, којој се прослеђују Скала-функција, повратни тип и типови аргумената функције. Пример креирања *udf*-а и његовог коришћења у трансформацији *withColumn* је приказан у коду 4.1.

```
val someFunction: (arg_type1, arg_type2) => ret_type_1 = (x:
   arg_type_1, y: arg_type_2) => {
  // function body
}
```

```
val udfRegistered = udf[ret_type_1, arg_type_1, arg_type_2](
   someFunction)

someDataFrame
  .withColumn("col_name", udfRegistered(arg1, arg2))
```

Кôд 4.1: Пример коришћења *udf*-а

Често се дешава да конструисан *udf* користи промењљиве које су креиране ван његовог тела, као у коду 4.2. Овакав приступ није добра пракса због тога што ће се низ *a* слати извршиоцима сваки пут када се ова операција иницира, што може знатно успорити извршавање.

```
val a: Array[Int] = Array(1, 2, 3, 4, 5)

val checkFunc: (Int) => Boolean = (x: Int) => {
  a.contains(x)
}

val udfRegistered = udf[Boolean, Int](checkFunc)
```

Кôд 4.2: Пример коришћења промењљиве дефинисане ван тела *udf*-а

Да би се избегло слање података сваки пут, могу се користити промењљиве *broadcast* [7]. Оне се кеширају у извршиоцима и увек су доступне, чиме се избегава беспотребно слање података. Промењљиве *broadcast* се креирају наредбом *broadcast*, објекта *SparkContext* и њиховој вредности се приступа преко кључне речи *value*. Ове промењљиве се могу користити само за читање података. У коду 4.3 се налази измењен кôд 4.2 тако да користи *broadcast*-промењљиву.

```
val a: Array[Int] = spark.sparkContext.broadcast(
  Array(1, 2, 3, 4, 5)
)

val checkFunc: (Int) => Boolean = (x: Int) => {
  a.value.contains(x)
}
```

```
val udfRegistered = udf[Boolean, Int](checkFunc)
```

Кôд 4.3: Пример коришћења промењиве *broadcast*

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

Поред различитог начина представљања података, постоји знатна разлика у перформансама између *RDD*-ја и *DataFrame*-а [14]. *RDD* се користи за програмирање ниског нивоа, пошто омогућава директан рад са партицијама. Међутим, приликом писања *RDD*-трансформација, програмер мора бити веома пажљив када и коју трансформацију примењује, због тога што редослед може значајно да утиче на перформансе.

Са друге стране, редослед примене трансформација *DataFrame*-а не утиче на брзину извршавања [14]. Разлог томе је што сваки *DataFrame* кôд пролази кроз аутоматски процес оптимизације, па ће добијени резултат увек бити најбржи могући. За оптимизацију је задужен процес који се зове *Catalyst*. Због перформанси, али и због једноставнијег интерфејса, *DataFrame* је скоро потпуно потиснуо *RDD* из употребе.

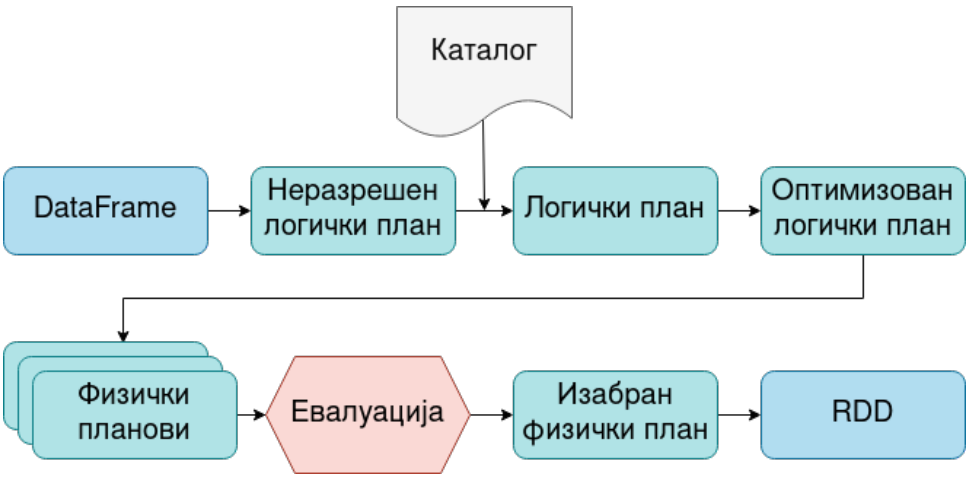
Планови извршавања *DataFrame*-а

Сваки *DataFrame* се приликом покретања прво преводи у *RDD*, након чега се извршава. Добијени *RDD* кôд ће увек имати оптимизован редослед примена трансформација. Да би се оптимизација успешно изршила, *Spark* током превођења генерише неколико планова извршавања.

Први план који се конструише од *DataFrame* кода је неразрешен логички план (енг. *unresolved logical plan*). Он представља трансформације које треба извршити над *DataFrame*-ом, али не садржи никакве информације о томе над којим колононама, као ни о томе где се подаци које *DataFrame* представља физички налазе [14]. Те информације се добијају уз помоћ каталога (енг. *catalog*) у коме се налазе информације о *DataFrame*-овима. Резултат примене каталога на неразрешен логички план је логички план (енг. *logical plan*). Анализом логичког плана и применом правила оптимизације на њега, оптимизатор *Catalyst* конструише оптимизован логички план (енг. *optimized logical plan*).

Након што се оптимизовани логички план успешно креира, *Spark* у односу на њега конструише неколико физичких планова (енг. *physical plan*)

[14]. Физички план дефинише на који начин и уз помоћ којих наредби ће се логички план извршити на кластеру. Сви физички планови се након тога евалуирају и бира се онај са најбољим перформансама. Он се преводи у *RDD*-трансформације и извршава на кластеру. Цео процес је приказан на слици 4.9.



Слика 4.9: Ток извршавања *DataFrame*-а

Измена броја партиција

За сваку апстракцију података, *Spark* иницијално креира одређен број партиција које она садржи. На пример, уколико се *DataFrame* креира читањем фајла, број партиција ће бити одређен или бројем партиција у фајлу, уколико фајл формат садржи такву информацију, или параметром *Spark*-конфигурације *spark.files.maxPartitionBytes* [9], који одређује максималну величину партиције. Подразумевана вредност параметра је $128MB$. Поред тога, параметар *spark.default.parallelism* [9] одређује колико партиција ће имати апстракција која је резултат *join* или трансформација *reduceByKey*. *DataFrame* који настаје од *RDD*-ја позивом функције *spark.createDataFrame* чији су аргументи *RDD* и шема, ће имати исти број партиција као *RDD* од ког настаје.

Број постојећих партиција је битан због тога што се у једном тренутку, на једном извршиоцу, обрађује тачно једна партиција. Уколико је број партиција мањи од броја извршилаца, постојаће извршиоци који неће обрађивати податке, што значи да искоришћеност кластера није добра. Уколико је број партиција велики, извршавање ће се успорити због тежег распоређивања *Spark*-задатака. Такође, операције *shuffle* постају доста скупље.

Из наведених разлога постоје трансформације које мењају број партиција. Те трансформације су *repartition* и *coalesce*. Друга се користи само у случају када је потребно смањити број партиција. Предност ове трансформације је избегавање операције *shuffle*, али не гарантује да ће подаци бити подељени подједнако. Са друге стране, *repartition* ће увек применити *shuffle* али податке по партицијама распоређује подједнако. *Repartition* може поделити податке на два начина. Први је прослеђивањем жељеног броја партиција, а други је навођењем жељене колоне *DataFrame*-а по чијим вредностима се извршава партиционисање.

Поред њих, постоји и партиционисање фајла приликом уписа. Извршава се функцијом *partitionBy* којој се наводе колоне по којима се извршава партиционисање. Фајл настао на овај начин групише редове које имају исте вредности у колонама по којима се врши партиционисање у заједничке поддиректоријуме. На пример, уколико *DataFrame* садржи колоне које представљају годину и месец, назване редом *year* и *month* и остале колоне назване *data*, приликом партиционисања при уписивању преко колона *year* и *month* ће настати фајл структура приказана у коду 4.4. Имена и вредности колона по којима се извршава партиционисање се налазе у називима поддиректоријума, док се подаци колона *data* налазе у крајњим фајловима.

```
|-- year=2022
|   |-- month=12
|   |   |-- part-00000-file-name.format
|   |   |-- part-00001-file-name.format
|   |-- month=11
|   |   |-- part-00000-file-name.format
|   |   |-- part-00001-file-name.format
|   |-- month=etc
|   |   |--etc
|-- year=2021
|   |-- month=12
|   |   |-- part-00000-file-name.format
|   |   |-- part-00001-file-name.format
|   |-- month=11
|   |   |-- part-00000-file-name.format
|   |   |-- part-00001-file-name.format
|   |-- month=etc
```

```
|    |    |--etc
|-- year=etc
```

Кôд 4.4: Пример структуре излазног директоријума који настаје партиционисањем

4.5 Остале компоненте *Spark*-а

Spark омогућава коришћење *SQL*-упита над подацима. Сваки *Spark SQL*-упит пролази кроз исти процес оптимизације као и *DataFrame* [14]. Једина разлика је у томе што се синтаксне грешке *SQL* кода појављују током извршавања програма, док се синтаксне грешке *DataFrame* кода појављују приликом компајлирања. У коду 4.5 је приказано када долази до појаве грешке приликом покретања еквивалентних *DataFrame* и *SQL* наредби са погрешно написаном речи *select*.

```
# Spark DataFrame
dataframe.slect()
>>> compilation error

# Spark SQL
spark.sql('slect * from dataframe')
>>> runtime error
```

Кôд 4.5: Извршавање *DataFrame* и *SQL* кодова са грешком у писању

Поред апстракција података *RDD* и *DataFrame*, постоји и *DataSet*, који је доступан само у језицима заснованим на Јавиној виртуелној машини, Скали и Јави [14]. Представља податке на исти начин као *DataFrame* и пролази кроз исти процес оптимизације. Разлика је у провери типова вредности унутар колона, која се код *DataFrame*-а дешава током извршавања програма, док се код *DataSet*-а провера типова ради за време компајлирања.

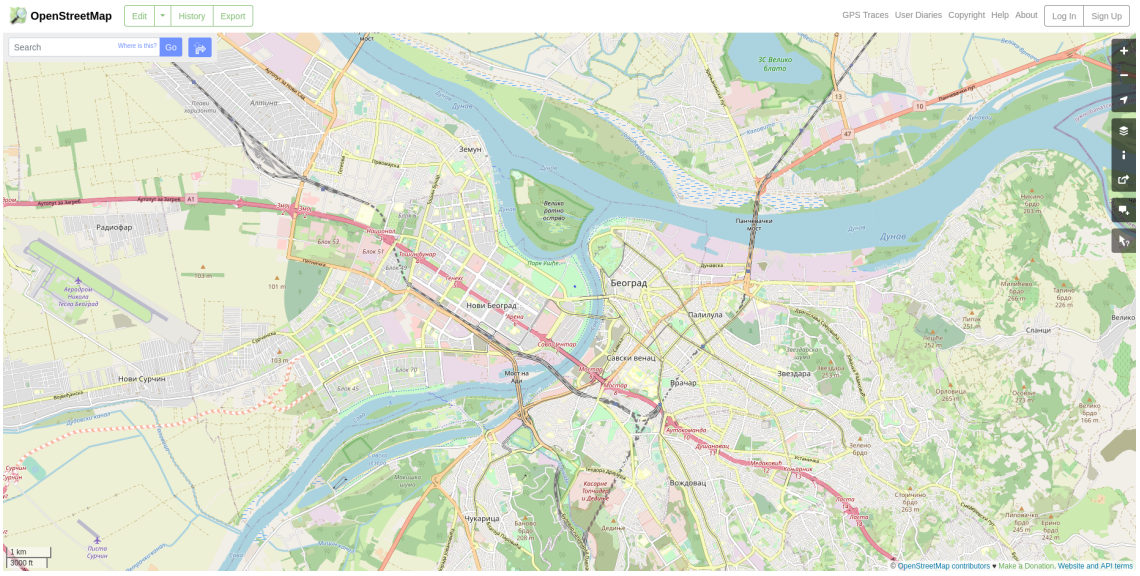
Уз помоћ *Spark*-а се могу конструисати модели машинског учења, преко библиотеке *Spark MLlib* [14]. Она се може користити за препроцесирање, тренирање модела и прављење предвиђања. *Spark* поседује и библиотеку за рад са графовима, *GraphX*, али се због њених разних недостатака, за обраду графова на кластеру често користе друга решења.

Spark се може користити и за операције над токовима података [37]. *Spark streaming* омогућава претплату на токове који настају из веб сокета (енг. *web socket*), локације у фајл систему или од стране *Apache Kafka*-е. Након претплате, на податке у току се могу примењивати исте трансформације као код *DataFrame*-а.

Глава 5

Скуп података *OpenStreetMap*

OpenStreetMap, скраћено *OSM*, је бесплатна мапа света која дозвољава приступ географским мапама, као и подацима које те мапе садрже [25]. Основна идеја овог пројекта је да заједница корисника развија и одржава мапе које представљају алтернативу већ постојећим мапама, попут оних које развија *Google* [28]. Пример мапе *OSM* на вебу је приказан на слици 5.1.



Слика 5.1: Приказ Београда у *OSM*-у

OSM је 2004. године покренуо Стив Коуст (енг. *Steve Coast*) са идејом креирања мапа за Уједињено Краљевство. У наредним годинама пројекат је постао глобалан и сада садржи податке целог света [25].

5.1 Елементи

За моделовање података физичког света у оквиру *OSM*-а се користе *OSM*-елементи [25]. Постоје три врсте елемената и то су чворови (енг. *nodes*), путање (енг. *ways*) и релације (енг. *relations*).

Сваки од елемената може имати придружену једну или више ознака (енг. *tag*) чија је улога да опишу елемент коме припадају. Елементи *OSM*-скупа се могу представити помоћу записа *XML* од којих је сваки одређен засебном *XML*-ознаком [26]. Сваки елемент у запису *XML* поседује атрибуте који га описују. Постоје одређени атрибути који се налазе у сваком елементу:

- ***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 видљив;
- ***version***, тренутна верзија елемента (почетна вредност је 1 и сваки пут када се изврши модификација елемента тај број се инкрементира);
- ***changeset***, идентификатор скупа промена у коме је елемент измењен.

Иако се подаци *OSM* могу представити помоћу записа *XML*, за обраду података је препоручен формат *PBF* [3]. У односу на *XML*, *PBF*-фајл заузима мање меморије и има боље перформансе за читање и писање података. *OSM*-фајлови формата *PBF* имају екстензију **.osm.pbf*.

Ознаке

Ознаке представљају опис *OSM*-елемента коме припадају. Сваки елемент може имати нула, једну или више ознака [25]. Чине је две вредности, кључ, који мора бити јединствен унутар елемента ког ознака описује, и вредност. У коду 5.1 је приказан пример ознака у формату *XML*. Кључ и вредност су означени редом атрибутима k и v. Приказане ознаке припадају примеру чвора из кода 5.2 и показују да чвор означава саобраћајни знак.

```
<tag k="name" v="Neu Broderstorf"/>
<tag k="traffic_sign" v="city_limit"/>
```

Кôд 5.1: Пример *OSM*-ознака у формату *XML*

5.2 Чворови

Чвор представља локацију на Земљиној површини и састоји се од две координате које представљају географску дужину и ширину [25]. Један чвор се може користити за дефиницију објекта на мапи, попут, на пример, клупе, статуе, хотела или ресторана.

У језику *XML* чворови су представљени *XML*-ознаком *node* унутар које су угњеждене *OSM*-ознаке које јој припадају. Кôд 5.2 представља један чвор записан у формату *XML*. Атрибути *lat* и *lon* представљају координате чвора на мапи. Чвор садржи две ознаке, које означавају да се на координатама чвора налази саобраћајни знак који представља улазак у насеље *Neu Broderstorf*.

```
<node id="1831881213" version="1" changeset="12370172" lat="
    54.0900666" lon="12.2539381" user="lafkor" uid="75625"
    visible="true" timestamp="2012-07-20T09:43:19Z">
  <tag k="name" v="Neu Broderstorf"/>
  <tag k="traffic_sign" v="city_limit"/>
 </node>
```

Кôд 5.2: Запис *XML OSM*-чвора који представља саобраћајни знак

5.3 Путање

Путање су уређене листе које садрже између 2 и 20000 чворова и представљају линеарне објекте на мапи, попут путева или река [25]. Такође, могу представљати и разне врсте површина, попут шума. У том случају су први и последњи елемент листе исти чвор.

Путање су у формату *XML* представљене листом идентификатора чворова које та путања садржи. Сваки чвор путање је записан *XML*-ознаком *nd* са атрибутом *ref* унутар ког се налази идентификатор чвора. Поред идентификатора чворова, путања може садржати и *OSM*-ознаке. *XML*-ознака која означава путању је *way*. У коду 5.3 је приказан пример путање која

представља улицу. Унутар *OSM*-ознаке путање је записано име улице, као и информација о томе да је улица једносмерна.

```
<way id="5090250" visible="true" timestamp="2009-01-19T19:07:25Z
    " 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>
```

Кôд 5.3: Запис *XML OSM*-путање која представља улицу

5.4 Релације

Релације су структуре које представљају однос између *OSM*-елемената [25]. Могу имати разна значења па су због тога описане *OSM*-ознакама. Обично, свака релација поседује *OSM*-ознаку која се зове *type* и свака друга ознака те релације се интерпретира на основу њене вредности.

У формату *XML*, релација се означава ознаком *relation* и садржи чланове релације и *OSM*-ознаке. Члан је одређен *XML*-ознаком *member* и садржи три атрибута:

- ***type***, *OSM*-тип члана, може бити *node*, *way* или *relation*;
- ***ref***, идентификатор елемента члана;
- ***role***, улога члана у релацији.

Репрезентација *XML* релације која представља аутобуску линију приказана је у коду 5.4. У овом примеру, *OSM*-чворови који припадају релацији представљају аутобуске станице. Поред чворова, релацији припада и једна путања, која приказује путању аутобуске линије. Ознаке релације приказују назив почетне и завршне локације линије, као и информације о превознику.

```
<relation id="13092746" visible="true" version="7" changeset="
    118825758" timestamp="2022-03-23T15:05:48Z" user="" uid="">
  <member type="node" ref="5690770815" role="stop"/>
  <member type="node" ref="5751940550" role="stop"/>
  ...
  <member type="node" ref="1764649495" role="stop"/>
  <member type="way" ref="96562914" role=""/>
  ...
  <member type="way" ref="928474550" role=""/>
  <tag k="from" v="Encre"/>
  <tag k="name" v="9-Montagnes de Guyane"/>
  <tag k="network" v="Agglo'bus"/>
  <tag k="not:network:wikidata" v="Q3537943"/>
  <tag k="operator" v="CACL"/>
  <tag k="ref" v="9"/>
  <tag k="route" v="bus"/>
  <tag k="to" v="Lyce Balata"/>
  <tag k="type" v="route"/>
</relation>
```

Кôд 5.4: Запис *XML OSM*-релације која представља аутобуску линију

Глава 6

Опис апликације *Geo-locator*

У овом поглављу ће детаљно бити описана израђена апликација за дистрибуирану обраду и графички приказ геопросторних података. Опис апликације укључује опис коришћених података, компоненти и технологија. Апликација је названа *Geo-locator* и налази се на адреси *https://github.com/davgav123/geo*.

Намена апликације је да филтрира геопросторне податке за државе Европе на такав начин да издвоји одређене битне локације, попут болница, апотека, ресторана и хотела. Издвојене локације за сваку државу се складиште и приказују на захтев корисника.

Део апликације за дистрибуирану обраду података је израђен у програмском језику Скала, коришћењем алата *Apache Spark*, док је за кориснички интерфејс коришћен језик *JavaScript*. Скуп геопросторних података који садржи локације које треба издвојити је *OpenStreetMap*. Из разлога што *OSM*-скуп садржи неколико десетина гигабајта података, за израду апликације је коришћен облак компаније *Amazon*.

6.1 Рачунарство у облаку

Kako је *OSM*-скуп података превелики да би се обрадио локално, у изради апликације је коришћен облак (енг. *cloud*). Извршавање на облаку (енг. *cloud computing*) [12] представља закупљивање хардвера и софтвера које нуди провајдер, на захтев, преко интернета, где се плаћање извршава по количини искоришћених ресурса. На пример, преко облака је могуће закупити базу података која ће преко интернета бити доступна одмах и плаћање ће се извршавати док год се та база података користи. Коришћење услуга облака је

приказано на слици 6.1.

Слика 6.1: Приказ коршћења услуга облака

У изради апликације коришћен је Амазонов облак (енг. *Amazon Web Services*), скраћено *AWS*. Хардвер и софтвер који се може закупити од *AWS*-а се назива сервис. Израђена апликација за дистрибуирану обраду података користи два сервиса, *S3* [11] и *EMR* [10].

Сервис *S3*, скраћено од *Simple Storage Service*, се користи за поуздано складиштење фајлова на *AWS*-у. Складиштени фајлови могу бити било ког формата и распоређени су у кофе (енг. *bucket*), које се могу посматрати слично као директоријуми у фајл систему рачунара. У апликацији се *S3* користи за складиштење *OSM*-фајлова над којима се врши обрада и за складиштење резултата обраде.

Сервис *EMR*, скраћено од *Elastic Map Reduce*, представља екосистем *Hadoop* на *AWS*-у. Састоји се од машина којима се може бирати снага процесора и количина меморије, зависно од потребе. На *EMR*-у се може инсталирати велики број апликација екосистема *Hadoop*, укључујући *Apache Yarn* и *Apache Spark*. Апликација користи овај сервис приликом дистрибуиране обраде података.

6.2 Подаци

У изради апликације су коришћена два скупа података. Први скуп представља *OSM*-податке Европе и у њему се налазе информације о географским локацијама. Изабрани *OSM*-скуп података је у формату *PBF* (енг. *Protocolbuffer Binary Format*) [3]. Пошто *Apache Spark* нема уграђену функционалност за читање података у формату *PBF*, подаци се морају пребацити у неки други формат, за који постоји једноставна интеграција са *Spark*-ом. Један од тих формата је формат отвореног кода, дизајниран за брзо читање и уписивање података, *Apache Parquet* [5]. Фајл формата *PBF* се може изменити у *parquet* коришћењем апликације *osm-parquetizer* [4]. Ова апликација прима *OSM*-скуп у формату *PBF* и као резултат има три фајла у формату *parquet*. Резултујући фајлови представљају одвојено *OSM*-чворове (са суфиксом *node*), путање (са суфиксом *way*) и релације (са суфиксом *relation*) почетног *PBF*-фајла. Како су за израду апликације *Geo-locator* потребне само локације са описима, које садрже *OSM*-чворови, релације и путање се могу одбацити. Укупна величина резултујућег скупа *OSM*-чворова који представљају Европу је око 60 гигабајта.

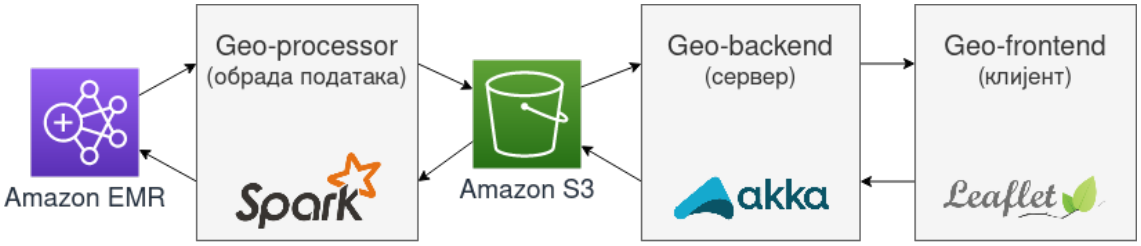
Други скуп, *GeoNames* [1] садржи информације о државама света, укључујући њихове границе, популацију, главни град, валуту, позивни број и слично. Границе држава скупа *GeoNames* су представљене полигонима којима су темена локације на географској мапи. Уколико се држава састоји из више целина, на пример острва, границе су представљене мултиполигоном чији су делови полигони који представљају једну целину државе. У супротном, границе су представљене једним полигоном. Скуп *GeoNames* се састоји из два фајла. Први, *shapes_all_low*, садржи границе држава, док други, *country_info*, садржи опште информације о државама, укључујући њихово име. Колона која их спаја представља јединствени идентификатор државе и назива се *geoNameId*.

6.3 Архитектура апликације

Апликација се састоји од три компоненте (слика 6.2). Прва, компонента за обраду података, *Geo-processor*, чита податке *OSM* и скупа *GeoNames* са локације на сервису *S3*, обрађује их помоћу Скале, *Spark*-а и сервиса *EMR*, и након тога резултате уписује назад на *S3*.

Друга компонента, названа *Geo-backend* представља сервер, који на захтев клијента чита обрађене податке и прослеђује их клијенту. Због високе цене обраде, подаци се не обрађују на сваки захтев клијента, већ се читају складиштени, већ обрађени подаци. Сервер апликације је написан у програмском језику Скала, коришћењем библиотеке *Akka* [30], уз помоћ које се креирају сервер и постојећи *http*-методи.

Последња компонента, клијент, названа *Geo-frontend*, приказује мапу света, прима упит корисника и исцртава жељене резултате. Страница клијента је приказана на слици 6.3. Написана је у програмском језику *JavaScript* и за исцртавање географске мапе и приказ резултата користи библиотеку *Leaflet* [2].

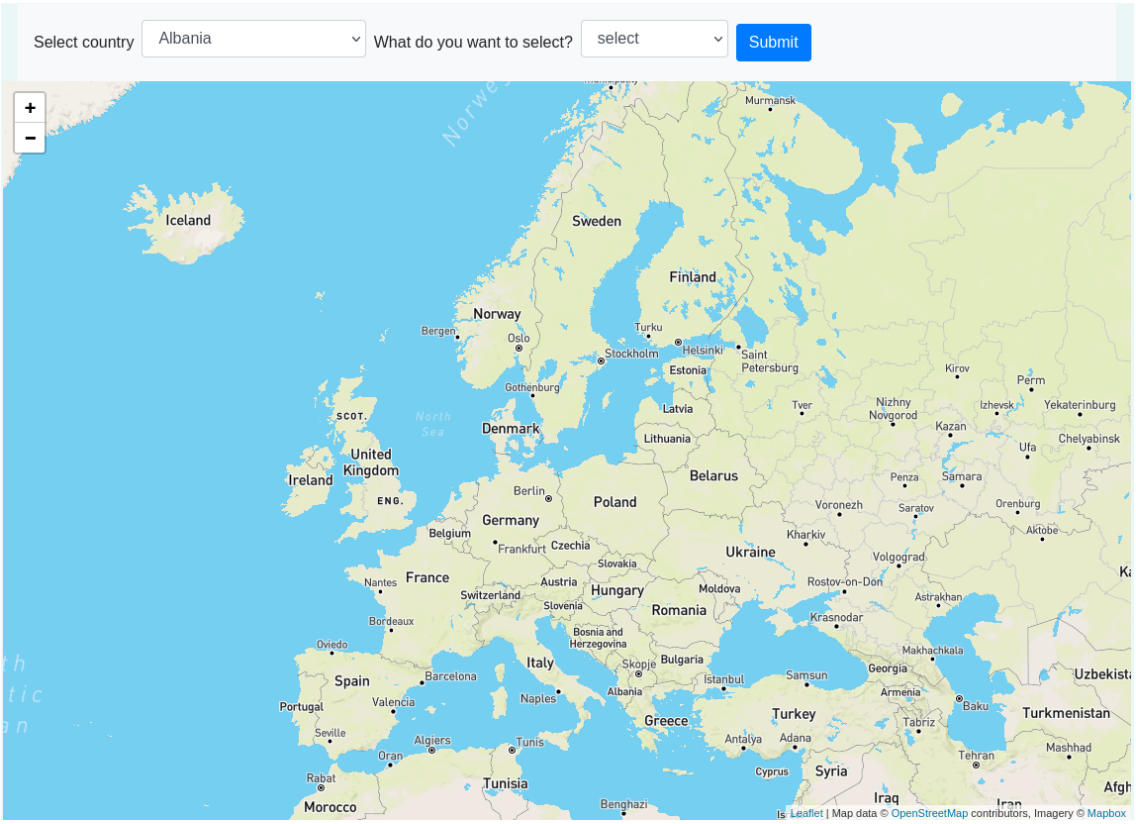


Слика 6.2: Компоненте апликације

6.4 Одређивање припадности локације држави

Проблем одређивања припадности локације држави је еквивалентан проблему одређивања припадности тачке полигону, где је локација тачка, а граница државе полигон или, за одређене државе, мултиполигон. Како је проблем припадности тачке полигону веома заступљен у рачунарској графици, постоји велики број алгоритама који га решавају. У овом раду су испробана три алгоритма и њихова имплементација се може пронаћи унутар класе *Polygon* компоненте *Geo-processor*. Ти алгоритми су:

- Алгоритам *Ray casting* [13];
- алгоритам заснован на сумирању углова [13];



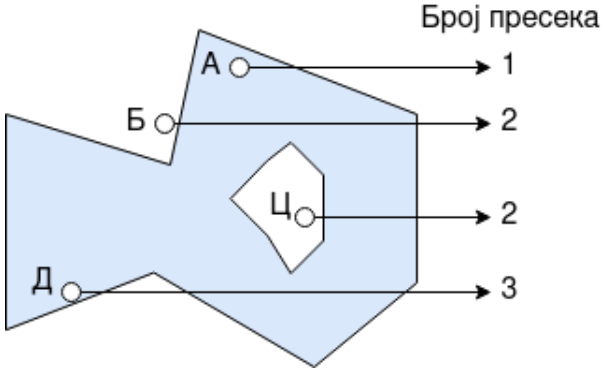
Слика 6.3: Клијентска страница апликације

- алгоритам доступан у оквиру Јава библиотеке *awt* [1], у класи *awt.Polygon* [33].

Алгоритам *Ray casting* исцртава хоризонталне дужи из жељене тачке, такве да јој се други крај налази изван полигона. Након тога се пребројава број пресека исцртане дужи са свим страницама полигона. Уколико је број пресека паран, тачка се налази ван полигона, а уколико је непаран, тачка се налази у полигону. Из примера алгоритма приказаног на слици 6.4 се може закључити да се тачке *А* и *Д* налазе унутар полигона, док су тачке *Б* и *Ц* изван. Недостатак алгоритма *Ray casting* је појава грешака приликом одређивања припадности тачака које се налазе близу ивица полигона.

Алгоритам заснован на сумирању углова сабира углове између жељене тачке и сваког пара темена полигона. Уколико је сума 2π онда се тачка налази унутар полигона, а уколико није онда се налази изван. Овај алгоритам

[1] имплементациони детаљи непознати



Слика 6.4: Алгоритам *Ray casting*

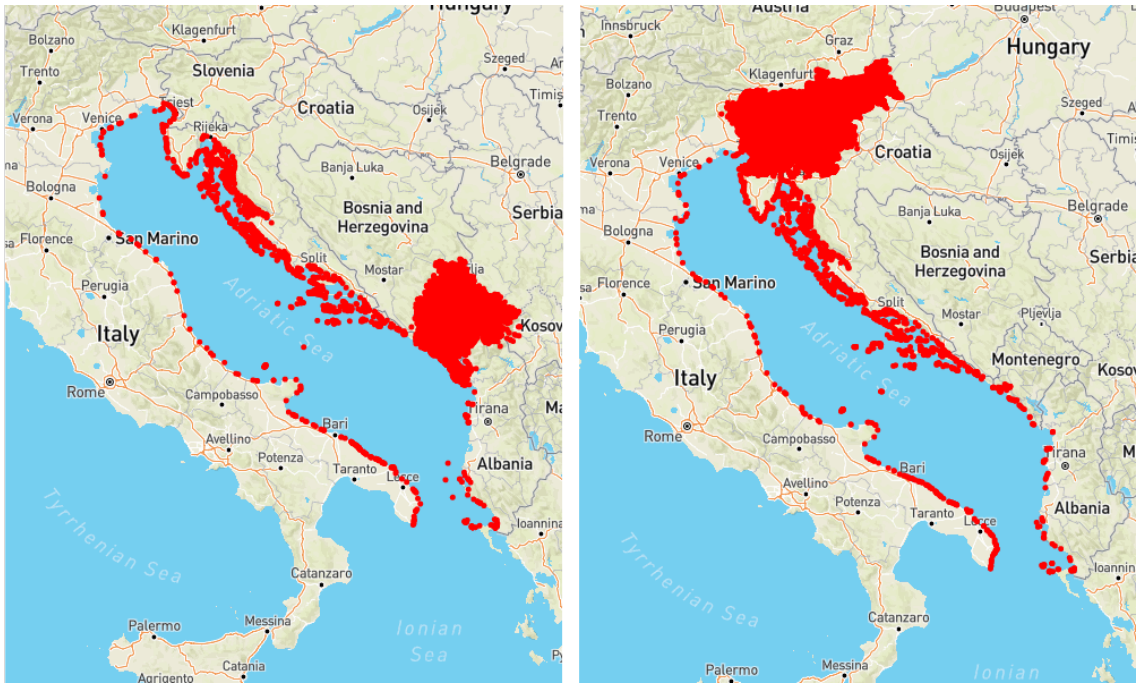
функционише за све врсте полигона. Као и *Ray casting*, овај алгоритам може погрешно одредити припадност за тачке близу ивица полигона.

Алгоритми су тестирани над *OSM*-скуповима који представљају Црну Гору и Словенију. Оба тест скупа садрже одређен број локација које не припадају државама које представљају, већ припадају њиховим суседима. То их чини повољним за тестирање, пошто се резултати алгоритама за припадност тачке полигону једноставно могу проверити исцртавањем на географској мапи. Локације из тест скупова су приказане црвеном бојом на слици 6.5.

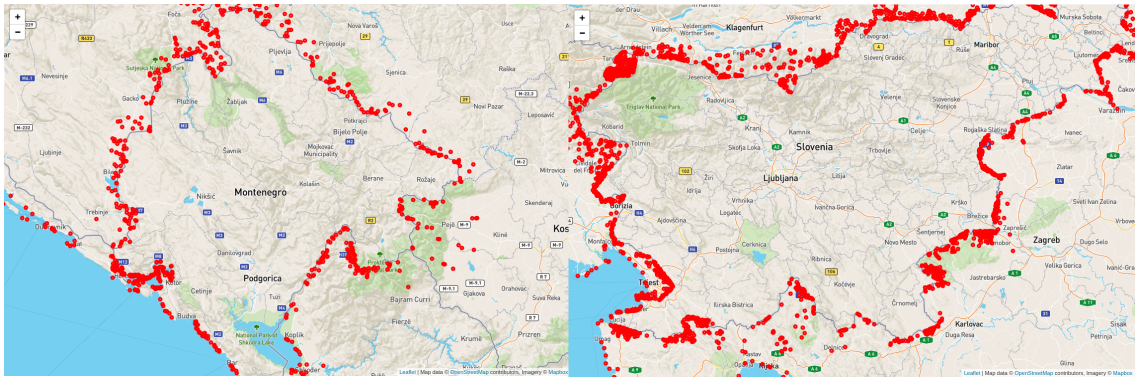
Добијени резултати за сваки алгоритам су углавном добри али постоје одређени недостаци. Сваки алгоритам је погрешно одредио припадност малом броју локација које се налазе близу обала мора. Поред тога, код сваког алгоритма постоје мање грешке за локације близу граница са другим државама. За разлику од друга два алгоритма, *Ray casting* је погрешно одредио припадност одређеном броју тачака на југоистоку Словеније. Због добијених резултата, у изради апликације је коришћен алгоритам заснован на сумирању углова. Резултати примене изабраног алгоритма на скупове за тестирање су приказани црвеном бојом, и на слици 6.6 представљају локације ван граница држава, док на слици 6.7 представљају локације унутар граница.

6.5 Обрада геопросторних података *Spark*-ом

Након трансформисања *OSM*-фајла из формата *PBF* у формат *parquet* и постављања *parquet* фајла на *S3*, може се почети са дистрибуираном обрадом података. Компонента апликације која обрађује податке је *Geo-processor*.



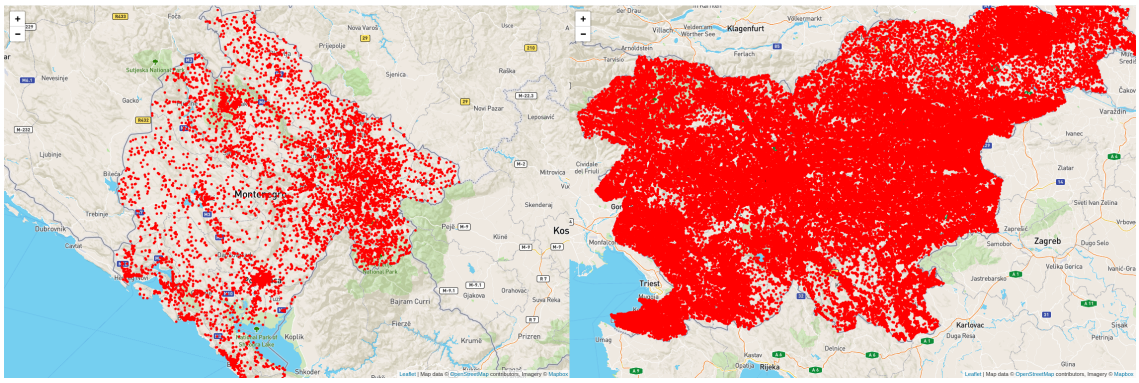
Слика 6.5: Локације из *OSM*-скупова Црне Горе и Словеније коришћених за тестирање



Слика 6.6: Резултат за локације изван држава за алгоритам заснован на сумирању углова

Написана је у верзији 2.12.12 програмског језика Скала коришћењем верзије 3.1.2 алата *Apache Spark*. Верзије су изабране због компатибилности. Изабрана верзија *Spark*-а је компатибилна са коришћеном верзијом *AWS* сервиса, док је верзија Скале компатибилна са верзијом *Spark*-а.

Подаци се обрађују на верзији 6.4.0 *EMR*-кластера. Одабрана верзија садржи инсталиране верзије 3.1.2 *Spark*-а и 3.2.1 *Hadoop*-а. Кластер се састоји



Слика 6.7: Резултат за локације унутар држава за алгоритам заснован на сумирању углова

од четири машине од којих је једна именски чвор, док су остале чворови података. Свака од машина садржи осам језгара процесора и тридесет два гигабајта радне меморије. *Spark* је конфигурисан тако да има шест извршилаца са по четири језгра процесора и шеснаест гигабајта меморије. Као менаџер ресурса се користи *Apache Yarn*. Трајање обраде података, које укључује и читање и упис обрађених података на *S3* је осам сати и тридесет седам минута.

Изворни кôд намењен за обраду података се налази унутар модула *geo* компоненте *Geo-processor*, у објекту *GeoDataProcessor*. Обрада се може поделити на два дела. Први чита податке и одређује која локација скупа припада којој држави, док друга за сваку државу филтрира жељену врсту локација након чега врши упис података. Оба дела су представљена у функцији *processGeoData*, која као аргументе прима иницијализовану инстанцу *Spark*-а, локацију скупа геопросторних података које треба обрадити и локацију у коју ће коначан резултат бити уписан. Приказ позива функције се налази у класи *Main* компоненте *Geo-processor* и приказан је у коду 6.1. Путање представљају локације на сервису *S3*, где је *geo-master-496542722941* име кофе, док остатак путање представља путању до фајла унутар кофе.

```
processGeoData(
  spark,
  "s3a://geo-master-496542722941/osm-data/europe/europe-latest.
   osm.pbf.node.parquet",
  "s3a://geo-master-496542722941/osm-data/countries/countries-
   data"
```

```
)
```

Кôд 6.1: Позивање функције која започиње обраду података

OSM-скуп чворова Европе, након примене апликације *osm-parquetizer*, поседује шему приказану у коду 6.2. Скуп се учитава унутар функције *getNodes* и из њега се трансформацијом *select* издвајају потребне информације, географска дужина и ширина, као и скуп *OSM*-ознака, у коме се налазе информације о томе шта се налази на координатама. Након тога се трансформацијом *where* подаци филтрирају тако што се одбацују сви редови којима су географска ширина и дужина *null* као и редови којима је скуп ознака празан. Имплементација функције *getNodes* се налази у коду 6.3

```
|-- id: long (nullable = true)
|-- version: integer (nullable = true)
|-- timestamp: long (nullable = true)
|-- changeset: long (nullable = true)
|-- uid: integer (nullable = true)
|-- user_sid: string (nullable = true)
|-- tags: array (nullable = true)
|    |-- element: struct (containsNull = true)
|    |    |-- key: string (nullable = true)
|    |    |-- value: string (nullable = true)
|-- latitude: double (nullable = true)
|-- longitude: double (nullable = true)
```

Кôд 6.2: Шема *OSM*-скупа чворова након примене апликације *osm-parquetizer*

```
def getNodes(spark: SparkSession, pathToCountryFile: String):
   DataFrame = {
  val nodes: DataFrame = spark
    .read
    .format("parquet")
    .load(pathToCountryFile)

  nodes.select(
    col("latitude"),
    col("longitude"),
    col("tags")
  )
```

```
    .where("latitude is not null and longitude is not null")
    .where(size(col("tags")) =!= 0)
}
```

Кôд 6.3: Функција која учитава податке и извршава иницијално филтрирање колона и редова

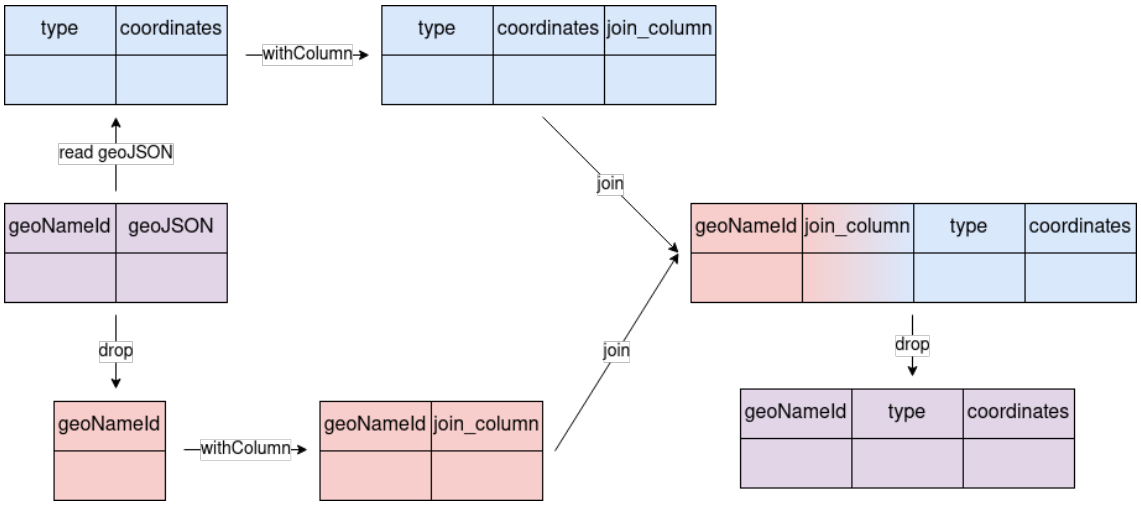
Након учитавања података, потребно је доделити одговарајућу државу за сваку локацију. Да би се тај корак извршио, морају се учитати и обрадити подаци скупа *GeoNames*, који се састоје из два фајла, сачувана на сервису *S3*. Први је *shapes_all_low* који је у формату *Tab Separated Values*, скраћено *TSV*, и садржи две колоне. Прва је *geoNameId*, док је друга *geoJSON*, у формату *JSON*. *JSON* садржи два кључа и први је ознака, названа *type*, која показује да ли су границе представљене једним полигоном (ознака *Polygon*) или са више полигона (ознака *MultiPolygon*). Други *JSON* кључ, назван *coordinates*, садржи листу координата које означавају полигоне. Пример редова овог скупа се налази у коду 6.4.

```
geoNameId geoJSON
49518     {"type":"Polygon","coordinates"
   :[[[29.96,-2.327],...,[29.438,-2.798]]]}
51537     {"type":"MultiPolygon","coordinates"
   :[[[[42.322,-0.659],...,[43.451,11.491]]]]}
```

Кôд 6.4: Упрошћени пример редова фајла *shapes_all_low*

Да би се информације из колоне *geoJSON* извукле у засебне колоне, потребно је издвојити ту колону, направити нови *DataFrame* од ње и додати колону *geoNameId*. Процес трансформације колоне *geoJSON* се налази у функцији *prepareBorderData* и приказан је у коду 6.5. Исти процес је приказан и на слици 6.8. Први корак је прочитати фајл у *DataFrame* и након тога од њега креирати нови, пребацивањем колоне *geoJSON* у засебан *DataFrame*. На такав начин се добија *DataFrame* са две колоне, *type* и *coordinates*. На њега се трансформацијом *withColumn* додаје још једна колона која ће се користити у спајању и која за сваки ред има вредност његовог редног броја. Након тога се из почетно прочитаног фајла *shapes_all_low*, издваја колона *geoNameId*, трансформацијом *drop*, којом се брише колона *geoJSON* и тако се конструише још један нови *DataFrame*. На њега се трансформацијом *withColumn* додаје колона за спајање, која за сваки ред има вредност његовог редног броја. На-

кон тога се два добијена *DataFrame*-а спајају преко колоне за спајање и колона за спајање се брише. Резултат је *DataFrame* са три колоне, *geoNameId*, *type* и *coordinates*.



Слика 6.8: Приказ трансформација над скупом *GeoNames* података који садржи границе

Добијени *DataFrame* поседује информације о границама, али не садржи имена држава, већ само њихов идентификатор. Имена се налазе у фајлу *country_info*. Након читања тог фајла, добијени *DataFrame* треба спојити са претходним и из спојеног *DataFrame*-а извући потребне колоне, *country* која представља име државе, *type* која означава да ли су границе полигон или мултиполигон и *coordinates* у којој се налазе координате граница. На крају се из добијеног скупа филтрирају државе које се не разматрају, што су у овом случају све државе које се не налазе у Европи. Имена европских државе се налазе у функцији *getEuropeanCountries*. Цео процес је приказан у коду 6.5 у коме се налази функција *prepareBorderData*.

```
private def prepareBorderData(spark: SparkSession): DataFrame =
  {
  val pathToBordersFile = "s3a://geo-master-496542722941/geo-
   names/shapes_all_low.txt"
  val pathToInfoFile = "s3a://geo-master-496542722941/geo-names/
   country_info.txt"

  val countryBordersRaw: DataFrame = spark
```

```
    .read
    .format("csv")
    .option("delimiter", "\t")
    .option("header", "true")
    .option("inferSchema", "true")
    .load(pathToBordersFile)

import spark.implicits._ // need this in order to cast json to
    string
val parsedJson: DataFrame = spark.read.json(countryBordersRaw.
  select(col("geoJSON")).as[String])
    .withColumn("join_column", monotonically_increasing_id())

val prepareBordersForJoin: DataFrame = countryBordersRaw
    .withColumn("join_column", monotonically_increasing_id())
    .drop(col("geoJSON"))

val countryBorders: DataFrame = prepareBordersForJoin
    .join(parsedJson, "join_column")
    .select(
      col("geoNameId"),
      col("type"),
      flatten(col("coordinates")).as("coordinates")
    )

    val countryToId: DataFrame = spark.read
    .format("csv")
    .option("delimiter", "\t")
    .option("header", "true")
    .option("inferSchema", "true")
    .load(pathToInfoFile)
    .select(col("geonameid").as("id"), col("Country"))

val borders: DataFrame = countryBorders
    .join(
      countryToId,
      countryBorders("geoNameId") === countryToId("id"),
```

```
      "inner"
    )
    .select(
      lower(col("Country")).as("country"),
      col("type").as("border_type"),
      col("coordinates").as("border_coordinates")
    )

  val selectedCountries: Array[String] = getEuropeanCountries
  borders
    .filter(col("country").isInCollection(selectedCountries))
}
```

Кôд 6.5: Функција која припрема *DataFrame* у коме се налазе информације о границама држава

За одређивање припадности локације држави, конструише се *udf*, приказан у коду 6.6. Улога *udf*-а је да за прослеђене географску ширину и дужину, одреди којој држави припадају. Повратна вредност је име државе, а уколико локација не припада ниједној држави, враћа се вредност „–“. Листа *iterableBorders* је *broadcast* променљива и садржи полигоне *GeoNames* скупа који представљају границе, док функција *isInsideBorder* за сваку локацију проверава којој држави припада тако што итерира кроз сваку границу и проверава припадност алгоритмом заснованим на сумирању углова.

```
val belongsToCountryFunction: (Double, Double) => String = (lat:
    Double, lon: Double) => {
  var belongs_to = "-"
  for ((country, borders) <- iterableBorders.value) {
    if (isInsideBorder(lat, lon, borders)) {
      belongs_to = country
    }
   }

  belongs_to
}
```

Кôд 6.6: Функција која додељује државу локацији

Процес додељивања државе локацији се извршава унутар функције приказане у коду 6.7. Након извршеног додељивања, из *DataFrame*-а се избацују све локације којима није додељена држава. На крају се извршава партиционисање података у односу на државу, трансформацијом *repartition*. Резултујући *DataFrame* ове функције садржи локацију, тачније географску ширину и дужину, *OSM*-ознаке, као и ознаку којој држави локација припада. Његова шема је приказана у коду 6.8.

```
def mapCoordinatesToCountry(spark: SparkSession, countryFilePath
    : String): DataFrame = {
  val iterableBorders = spark.sparkContext.broadcast(
    makeBordersIterable(prepareBorderData(spark))
  )

  val belongsToCountryFunction: (Double, Double) => String = (
   lat: Double, lon: Double) => {
    var belongs_to = "-"
    for ((country, borders) <- iterableBorders.value) {
      if (isInsideBorder(lat, lon, borders)) {
        belongs_to = country
      }
    }

    belongs_to
  }

  val belongsToCountry = udf[String, Double, Double](
   belongsToCountryFunction)

  val osmData = getNodes(spark, countryFilePath)
    .withColumn(
      "country",
      belongsToCountry(
        col("latitude"),
        col("longitude")
      )
    )
```

```
  osmData
    .filter(col("country") =!= lit("-"))
    .repartition(col("country"))
}
```

Кôд 6.7: Функција *mapCoordinatesToCountry*

```
|-- latitude: double (nullable = true)
|-- longitude: double (nullable = true)
|-- tags: array (nullable = true)
|    |-- element: struct (containsNull = true)
|    |    |-- key: binary (nullable = true)
|    |    |-- value: binary (nullable = true)
|-- country: string (nullable = true)
```

Кôд 6.8: Шема *DataFrame*-а након додељивања држава локацијама

Након одређивања припадности, потребно је за сваку државу изабрати жељену врсту локација. Филтрирање се извршава у односу на колону *tags* *DataFrame*-а. Пример садржаја колоне *tags* се налази у коду 6.9. Састоји се од низа кључ-вредност парова, од којих су за одабир жељених врста локације потребне само вредности. Оне се могу издвојити трансформацијом *withColumn* и применом функције *expr*, која користи функцију *transform* за трансформисање кључ-вредност парова (кôд 6.10).

```
+--------------------------------------------------------+
|tags                                                    |
+--------------------------------------------------------+
|[{access, yes}, {amenity, parking}, {parking, lane}]|
|[{amenity, restaurant}, {outdoor_seating, yes}]      |
|[{name, The Tesla Art Hostel}, {tourism, hostel}]    |
|[{amenity, hospital}, {healthcare, hospital}]        |
|[{name, The Black Turtle Pub I}, {amenity, pub}      |
|[{name, Picerija}, {amenity, cafe}]                  |
+--------------------------------------------------------+
```

Кôд 6.9: Пример вредности колоне *tags*

```
countryDF
  .withColumn(
    "tag_values",
```

```
    expr("transform(tags, x -> x.value)").cast(ArrayType(
   StringType))
  )
```

Кôд 6.10: Конструкција низа вредности колоне *tags*

Следећи корак је филтрирање добијеног низа вредности ознака. Слично као код одређивања припадности локације држави, за филтрирање жељених локација се користи *udf*, приказан у коду 6.11. Функционише тако што тражи пресек између жељених врста локација које се налазе у *broadcast* променљивој *conditions* и низа вредности ознака. Уколико пресек не постоји, функција враћа вредност „*x*“, а уколико постоји, функција враћа врсту локације. Резултат је нова колона названа *condition*. Након одређивања врсте локације, трансформацијом *filter* се уклањају сви редови који садрже вредност „*x*“. Све операције за филтрирање података се налазе унутар функције *applyFilters* приказане у коду 6.12.

```
val filterTagValues: Array[String] => String = (tagValues: Array
   [String]) => {
  val intersection = tagValues.toSet.intersect(conditions.value)
  if (intersection.isEmpty) {
    "x"
  } else {
    intersection.head
   }
}
```

Кôд 6.11: Функција која одређује да ли је локација жељеног типа

```
def applyFilters(spark: SparkSession, countryDF: DataFrame):
   DataFrame = {
  val conditions = spark.sparkContext.broadcast(
    Set(
      "hospital", "pharmacy", "hotel",
      "hostel", "bar", "cafe", "pub",
      "nightclub", "restaurant", "parking"
  ))

  val filterTagValues: Array[String] => String = (tagValues:
   Array[String]) => {
```

```
    val intersection = tagValues.toSet.intersect(conditions.
  value)
    if (intersection.isEmpty) {
      "x"
    } else {
      intersection.head
    }
  }

  val filterFunc = udf[String, Array[String]](filterTagValues)

  countryDF
    .withColumn(
      "tag_values",
      expr("transform(tags, x -> x.value)").cast(ArrayType(
  StringType))
    )
    .withColumn(
      "condition", filterFunc(col("tag_values"))
    )
    .filter(col("condition") =!= lit("x"))
}
```

Кôд 6.12: Функција *applyFilters*

Након филтрирања, добијене податке треба сачувати. Чување се извршава у функцији *processGeoData* (кôд 6.13), позваној у коду 6.1. Из података се прво извлаче жељене колоне, а затим се подаци чувају у формату *parquet*. Пре чувања, подаци се партиционишу по колонама *country* и *condition* функцијом *partitionBy*. Пример структуре излазног директоријума за државе Шпанију и Португал и за локације болница и хотела се налази у коду 6.14. Географске ширине и дужине се налазе у крајњим *parquet* фајловима, док се информације о држави и жељеној локацији налазе у именима директоријума који садрже *parquet* фајлове. Након чувања, подаци су спремни и могу се користити за приказивање на географској мапи.

```
def processGeoData(spark: SparkSession, pathToDataFile: String,
    pathToOutFile: String): Unit = {
  val mapped = mapCoordinatesToCountry(spark, pathToDataFile)
```

```
  applyFilters(mapped)
    .select(
      col("latitude"),
      col("longitude"),
      col("country"),
      col("condition")
    )
    .write
    .partitionBy("country", "condition")
    .mode(SaveMode.Overwrite)
    .parquet(pathToOutFile)
}
```

Кôд 6.13: Функција *processGeoData*

```
|-- country=portugal
|   |-- condition=hospital
|   |   |-- part-00000-c55ece3af5a5.c000.snappy.parquet
|   |   |-- part-00001-c55ece3af5a5.c000.snappy.parquet
|   |-- condition=hotel
|   |   |-- part-00000-c55ece3af5a5.c000.snappy.parquet
|   |   |-- part-00001-c55ece3af5a5.c000.snappy.parquet
|-- country=spain
    |-- condition=hospital
    |   |-- part-00000-f558ba92bda6.c000.snappy.parquet
    |   |-- part-00001-f558ba92bda6.c000.snappy.parquet
    |-- condition=hotel
        |-- part-00000-f558ba92bda6.c000.snappy.parquet
        |-- part-00001-f558ba92bda6.c000.snappy.parquet
```

Кôд 6.14: Структура директоријума партиционисаног по државама и врстама локација

6.6 Сервер апликације

Улога сервера апликације је да покрене сервер, прима захтеве *GET* клијента, обради их и пошаље резултат. Креирање сервера и обрада захтева

GET је имплементирана преко модула *http* библиотеке *Akka*, док се обрада извршава преко *Spark*-а. Креирање захтева *GET* је приказано у коду 6.15. Извршава се преко функције *path*, чији је аргумент ниска која означава путању до захтева. Параметри захтева се задају функцијом *parameters* док је повратна вредност функције одговор на захтев. У креираној функцији одговор је у формату *JSON*. Улога *Spark*-а у функцији *readData* је да прочита жељене локације креиране од стране компоненте *Geo-processor* и пребаци их у низ, који се као одговор шаље клијенту.

```
val route: Route = path("selection") {
  concat {
    cors() {
      get {
        parameters('country.as[String], 'param.as[String]) { (
   country, param) =>
          complete {
            val coords = readData(spark, country, param)
            HttpEntity(ContentTypes.`application/json`, s"""${
   coords.toJson}""")
          }
        }
      }
    }
  }
}

def readData(spark: SparkSession, country: String, cond: String)
   : Array[Array[Double]] = {
  val dataPath = "/path/to/data/dir/"
  val countryFilePath = dataPath + s"country=$country/condition=
   $cond"

  spark
    .read
    .parquet(countryFilePath)
    .collect()
    .map(row => Array(row(0).asInstanceOf[Double], row(1).
   asInstanceOf[Double]))
```

```
}
```

Кôд 6.15: Имплементација функције која прима и обрађује захтев *GET*

На крају се направљена путања везује за инстанцу сервера, функцијом *bind*. Сервер се налази на локацији *localhost*, на порту *8080* и покреће се функцијом *newServerAt*, објекта *Http*, библиотеке *Akka* [31]. Имплементација сервера је приказана у коду 6.16.

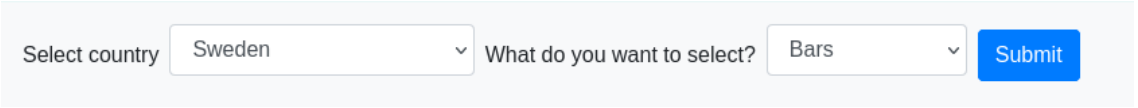
```
val bindingFuture = Http().newServerAt("localhost", 8080)
  .bind(route)

println(s"Server is now online\nPress RETURN to stop...")
StdIn.readLine()
bindingFuture
  .flatMap(_.unbind())
  .onComplete(_ => {
     spark.close()
     system.terminate()
  })
```

Кôд 6.16: Имплементација сервера

6.7 Клијент апликације

Клијент шаље захтеве *GET* серверу и приказује резултат на географској мапи. Слање захтева се извршава уносом жељених параметара и притиском на дугме *Submit* (слика 6.9). Притиском на дугме се креира захтев *http* ка адреси сервера. За слање захтева се користи objekat *XMLHttpRequest*. Када сервер пошаље резултате клијенту, он исцртава жељене локације црвеном бојом на мапи.

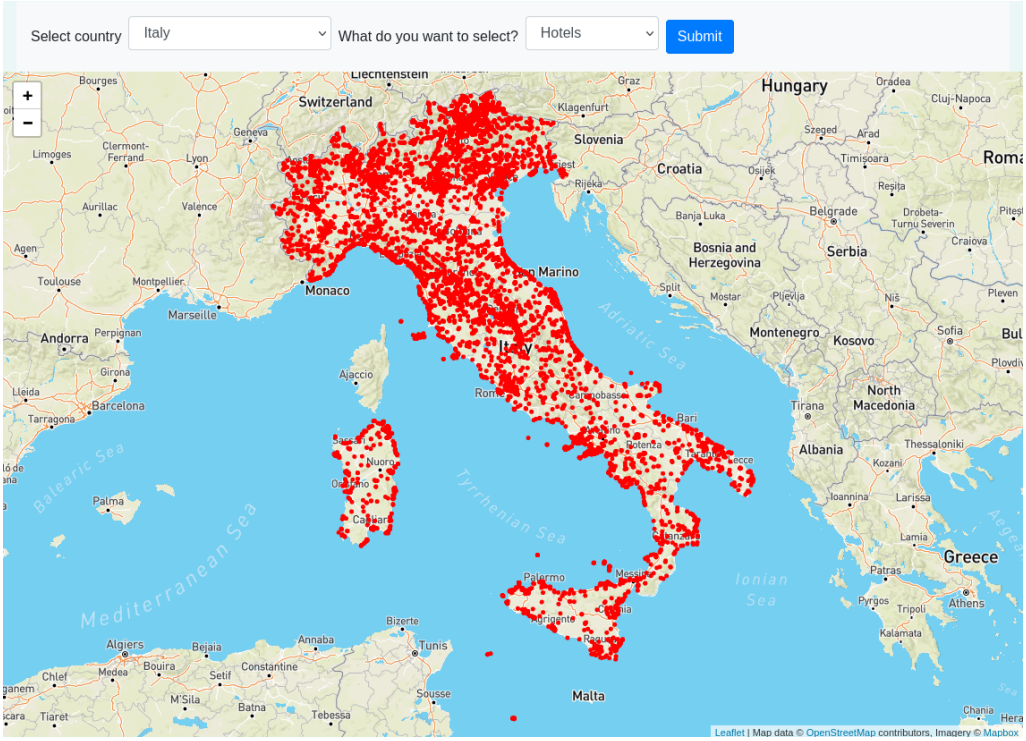


Слика 6.9: Место за унос и слање захтева ка серверу

За исцртавање мапе се користи библиотека *Leaflet*. Иницијално се приказује мапа целе Европе, али се при исцртавању жељених локација мапа увеличава на приказ изабране државе.

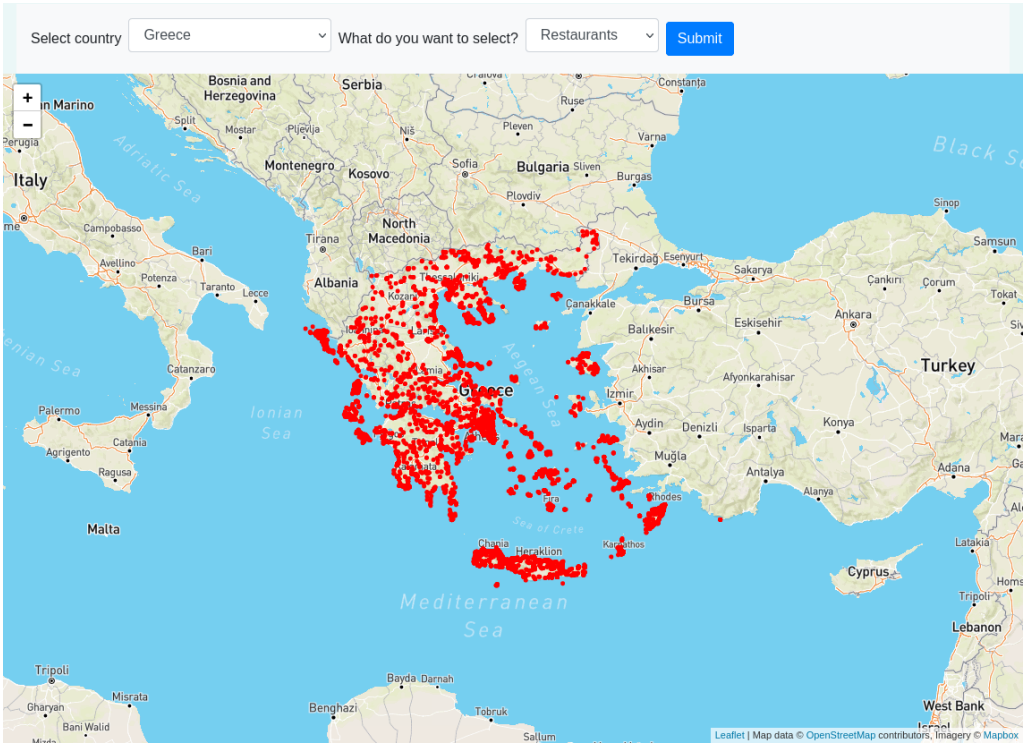
6.8 Приказ резултата

У овом одељку су приказани примери крајњег резултата рада апликације. У примерима на сликама 6.10, 6.11, 6.12 и 6.13 је приказана клијентска страница апликације са исцртаним локацијама. Приказане државе и локације су изабране да би се илустровале различите опције које корисник може да изабере.

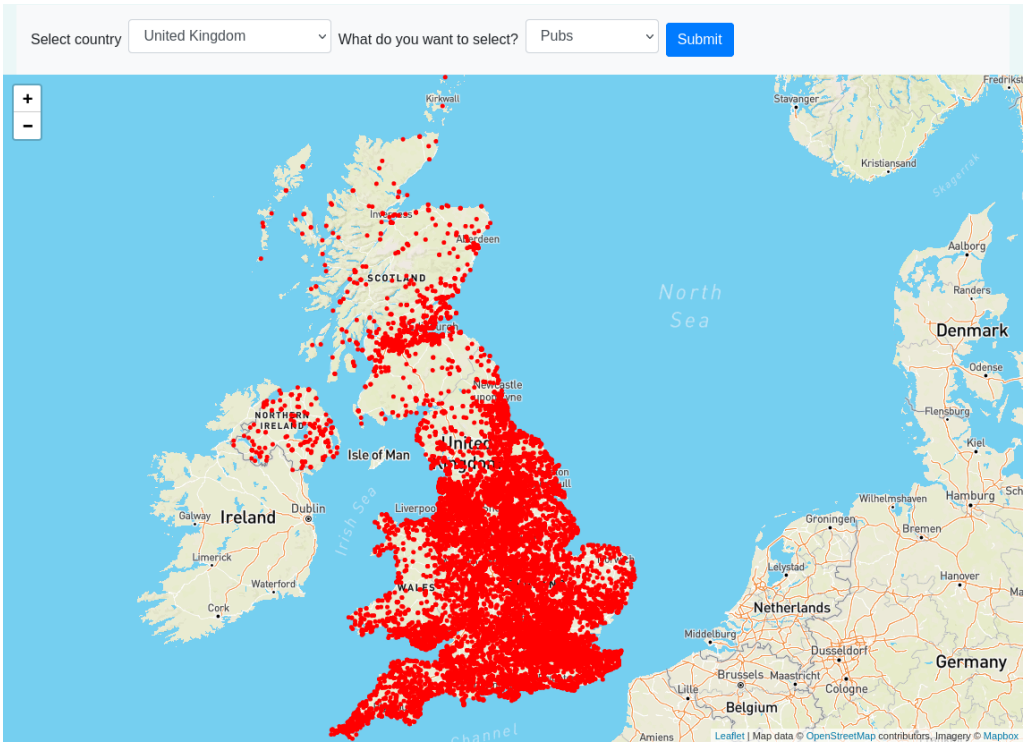


Слика 6.10: Хотели у Италији

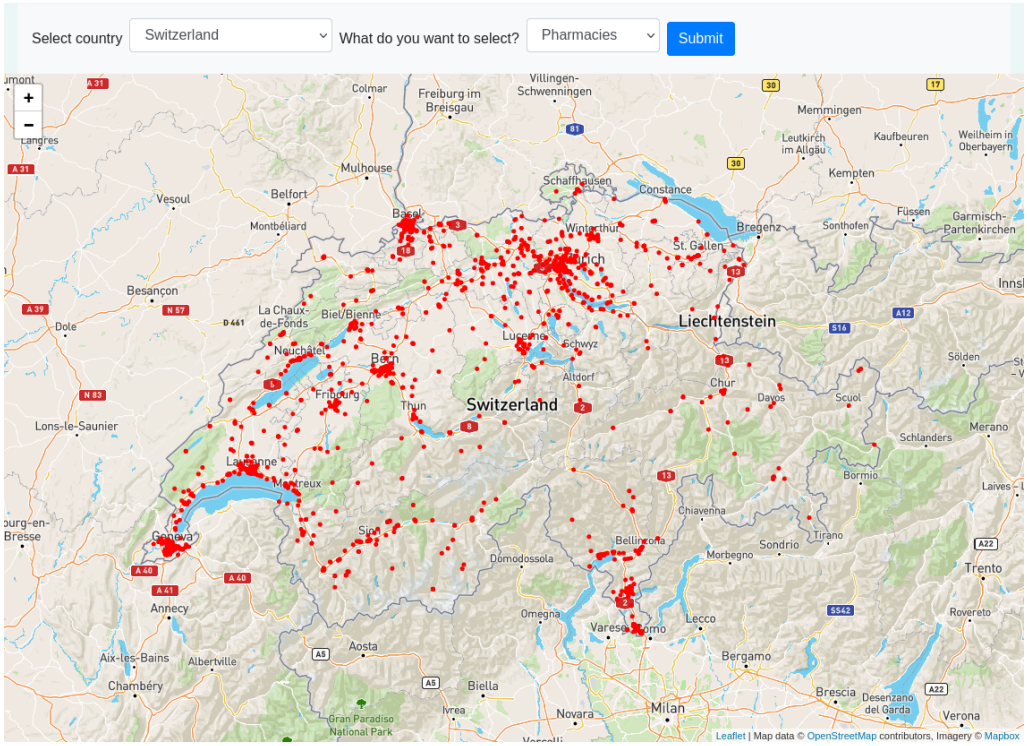
Одређивање припадности локација држави је било успешно за већину држава Европе. Међутим, како алгоритми за одређивање припадности тачке полигону нису савршени, дошло је до појаве грешака код мањих држава. Алгоритам је потпуно занемарио Сан Марино, и доделио све тачке које му припадају Италији (слика 6.14). Такође, појављују се грешке код државе Монако, чији је велики број тачака припојен Француској. Припадност локација за остале мање државе Европе (Луксембург, Малта, Андора и Лихтенштајн) је одређена без грешака.



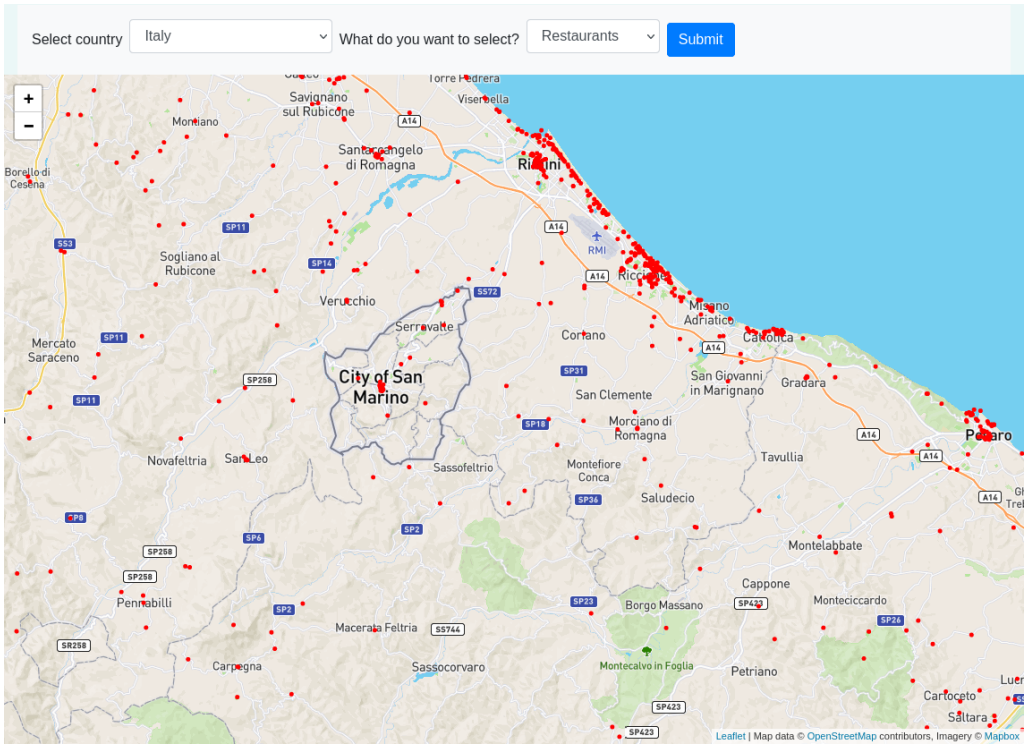
Слика 6.11: Ресторани у Грчкој



Слика 6.12: Пабови у Уједињеном Краљевству



Слика 6.13: Апотеке у Швајцарској



Слика 6.14: Сан Марино када се одабере приказ ресторана у Италији

Глава 7

Закључак

У раду су описане особине и основни принципи програмског језика Скала. Уз опис су приложени и конкретни примери. У овом програмском језику је написана апликација која је циљ овог рада али и *Apache Spark*, који се користи за дистрибуирану обраду података. Описани су основни концепти *Apache Spark*-а, попут архитектуре, партиционисања података, кеширања, трансформација и акција. Поред концепата, детаљније су описане две компоненте *Spark*-а за моделовање података *RDD* и *DataFrame*.

Приказана је и мотивација за коришћење дистрибуираних система као и њихова организација. Описан је дистрибуирани фајл систем *HDSF*, централни део екосистема *Hadoop* ког поред *HDFS*-а чине и остале апликације разних намена, попут *Apache Yarn*-а, *Flume*-а, *Hive*-а и *Pig*-а. Приказана је и прва парадигма за дистрибуирану обраду података, *MapReduce*-а, као и њени недостаци.

Имплементирана је апликација за дистрибуирану обраду скупа геопросторних података *OpenStreetMap*. Обрада се извршава на екосистему *Hadoop* имплементираном од стране *AWS*-а, сервису *EMR*. За обраду података се користе *Apache Spark* и програмски језик Скала. Намена апликације је да филтрира геопросторне податке тако што ће за сваку државу европе издвојити локације здравствених објеката, хотела, хостела, барова, ресторана и других битних туристичких локација које се унутар ње налазе, да би се те локације касније приказале на географској мапи. За приказ локација на географској мапи се користи језик *JavaScript* и библиотека *Leaflet*.

Апликацијом је постигнут жељени резултат, иако се појављују грешке за локације малих држава Европе, Сан Марина и Монака. Време извршавања

обраде података је задовољавајуће, али је потребно истражити да ли је могуће побољшање перформанси. Број понуђених опција за жељене локације садржи десет врста локација, али се једноставно може проширити, на пример додавањем религијских објеката, музеја и других туристичких атракција. Клијент и сервер компоненте су једноставне и имају добре перформансе.

Постоји неколико начина на које се апликација може унапредити. На пример, поред филтрирања локација по државама, филтрирање се може извршити и по регијама држава или градовима. Такође, могуће је и пратити тренутну локацију корисника и на основу ње приказивати филтриране локације. Да би подаци увек били ажурни потребно је направити аутоматизован систем који ће реаговати на објављивање сваке нове верзије скупа *OpenStreetMap* и поново га филтрирати. Такође, потребно је истражити да ли је могуће унапређење перформанси коришћењем других технологија за дистрибуирану обраду података уместо *Spark*-а, попут *Apache Sedona*-е [6].

Библиографија

[1] GeoNames. online at: https://www.geonames.org/.

[2] Leaflet. online at: https://leafletjs.com/.

[3] PBF Format. online: https://wiki.openstreetmap.org/wiki/PBF_Format.

[4] Adrianulbona. Osm parquetizer. online at: https://wiki.openstreetmap.org/wiki/Osm-parquetizer.

[5] Apache Foundation. Apache Parquet. online at: https://parquet.apache.org/.

[6] Apache Foundation. Apache Sedona. oneline at: https://sedona.apache.org/.

[7] Apache Foundation. Broadcast.

[8] Apache Foundation. Built-in Functions. online at: https://spark.apache.org/docs/latest/api/sql/index.html.

[9] Apache Foundation. Spark Configuration. online at: https://spark.apache.org/docs/latest/configuration.html.

[10] AWS. Amazon EMR. online at: https://aws.amazon.com/emr/.

[11] AWS. Amazon S3. online at: https://aws.amazon.com/s3/.

[12] AWS. What is cloud computing? online at: https://aws.amazon.com/what-is-cloud-computing/.

[13] Paul Bourke. Determining if a point lies on the interior of a polygon. 1997.

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

[15] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified Data Processing on Large Clusters . 2004. online at: https://research.google/pubs/pub62/.

[16] Apache foundation. Apache Flink. online at: https://flink.apache.org/.

[17] Apache foundation. Apache Flume. online at: https://flume.apache.org/.

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

[19] Apache foundation. Apache Kafka. online at: https://kafka.apache.org/.

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

[21] Apache foundation. Apache Spark. online at: https://spark.apache.org/.

[22] Apache foundation. Apache Yarn. online at: https://hadoop.apache.org/docs/stable/hadoop-yarn/hadoop-yarn-site/YARN.html.

[23] Apache foundation. Apache Zookeeper. online at: https://zookeeper.apache.org/.

[24] Apache foundation. HDFS Architecture Guide. online at: https://hadoop.apache.org/docs/.

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

[26] OpenStreetMap Foundation. OSM XML. online at: https://wiki.openstreetmap.org/wiki/OSM_XML.

[27] Presto foundation. Presto. online at: https://prestodb.io/.

[28] Google. Google Maps. online at: https://www.google.com/maps/.

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

[30] Lightbend. akka. online at: https://akka.io/.

[31] Lightbend. Introduction to akka Http. online at: https://doc.akka.io/docs/akka-http/current/introduction.html.

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

[33] Oracle. java.awt.Polygon. online at: https://docs.oracle.com/javase/7/docs/api/java/awt/Polygon.html.

[34] Howard Gobioff Sanjay Ghemawat and Shun-Tak Leung. The Google File System. 2003. online at: https://research.google/pubs/pub51/.

[35] Apache Spark. RDD Programming Guide. online at: https://spark.apache.org/docs/latest/rdd-programming-guide.html.

[36] Apache Spark. Spark Core. online at: https://spark.apache.org/docs/latest/api/python/reference/pyspark.html.

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

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

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

Биографија аутора

Давид Гавриловић, рођен је 29.10.1996. у Чачку. Смер Информатика на Математичком факултету Универзитета у Београду је уписао 2015. године, а завршио 2019. Након тога је на истом факултету уписао мастер студије информатике. У новембру 2020. године је почео са радом у компанији *Grid Dynamics* и од тад ради као *Data Engineer*. Тренутно ради у тиму који развија и одржава инфраструктуру и сервисе које користе *data science* и *data analyst* тимови. Интересује се за обраду великих података, машинско учење и рад са геопросторним подацима.