UNIVERSITATEA DIN BUCUREȘTI
FACULTATEA DE MATEMATICĂ ȘI INFORMATICĂ
SPECIALIZAREA INFORMATICĂ

LUCRARE DE LICENȚĂ

Sisteme de analitice bazate pe big data cu aplicații în managementul traficului

Coordonator științific:
Conf. Dr. Hacic-Cristian K. Kevorchian

Absolvent:
Roman Gabriel-Marian

București, februarie 2020

Introducere

Contextul lucrării

O mare parte dintre locuitorii marilor orașe depind de autovehiculul personal pentru a-și desfășura activitățile de zi cu zi. Traficul îngreunat și problemele de circulație sunt fenomene întâlnite frecvent, multe dintre acestea datorate comutei zilnice până la locul de muncă.

Conform unui studiu realizat de "Gândul", 1 din 4 cetățeni ai Bucureștiul conduc mașina personală zi de zi. 40% dintre cetățenii capitalei circulă cel puțin săptămânal cu mașina, chiar dacă nu toți în calitate de șoferi. De asemenea, în intervalul unei săptămâni obișnuite 10% din populația capitalei călătorește cu taxiul sau cu un serviciu de transport, cum ar fi Uber, cel puțin o dată.

Un alt studiu realizat de către publicația "Digi24" din anul 2019 atestă că un conducător auto irosește zilnic câte 57 de minute datorită traficului în București. Conform acestui studiu Bucureștiul este al 5-lea cel mai aglomerat oraș de pe glob. Fiind depășit doar de Jakarta, Bangkok, Chongqing și Mexico City.

De asemenea, din graficele actualizate în timp real de către aplicația de trafic Waze reiese că în timpul orelor de vârf, intervalele 7 - 11 și 17-20, aplicația are un număr constant de 20 de mii de utilizatori.

Scopul aplicației

Aplicația are ca scop analizarea datelor provenite din trafic și determinarea punctele din oraș în care există impedimente de circulație pe baza acestora. Astfel aplicația are două funcționalități. Prima este posibilitatea de a trimite alerte în timp real către utilizatori, permițând acestora să evite zonele în care se circulă cu dificultate în cazul în care sunt simpli participanți la trafic, sau poate fi folosită de către membri ai administrației municipale pentru a trimite în mod dinamic agenți rutieri pentru a fluidiza traficul în aceste zone.

A doua funcționalitate este cea de a crea un “heatMap” al orașului sau a unei secțiuni a acestuia, care va partiționa aria dată și apoi va prezenta intensitatea activității în trafic din zona respectivă pentru o perioadă de timp configurabilă, spre exemplu ultima lună. Pe baza acestora, utilizatorii pot evita preventiv zonele ce sunt predispuse la îngreunarea traficului atunci când decid traseul spre și dinspre locul de muncă.

Aplicația folosește tool-uri dedicate pentru big data întrucât informațiile referitoare la trafic sunt într-un număr foarte mare, updatate constant, iar procesarea lor în timp real și analizarea tuturor datelor primite de-a lungul unei perioade întinse de timp sunt procese foarte costisitoare dacă nu sunt optimizate. Din acest motiv aplicația se bazează pe motorul Spark pentru procesarea datelor în paralel, în timp real. Datele voi fi stocate în Kafka, unde pot fi citite și scrise în manieră **streaming**, și de asemenea Kafka oferă o performanță constantă în ce privește producerea și consumarea datelor, viteza de executare a operațiilor fiind independentă de numărul recordurilor.

Aplicația este scrisă în limbajul Scala, deoarece acesta este compatibil cu Spark și interoperabil cu Java, astfel că poate folosi driverul Kafka pentru Java. Scala este de asemenea un limbaj modern, scalabil, ce prin îmbinarea paradigmei programării orientate pe obiecte și pe cea a programării funcționale oferă flexibilitate și posibilitatea de a scrie cod compact, ușor de urmărit.

Pe lângă modulele de analiză, aplicația cuprinde un simulator de trafic rutier. Acesta creează un număr configurabil de autovehicule “prop”, care vor mima șofatul, iar cumulat vor crea iluzia de trafic rutier. Generatorul de trafic este util deoarece permite crearea instrumentelor de analiză și statistică în lipsa unei surse veridice de date. De asemenea este util pentru testarea de cazuri specifice sau dezvoltarea de funcționalități cu aplicații restrânse deoarece este configurabil și poate oferi un mediu controlat pentru testare.

Partea 1

Stiva de tehnologii folosite

1.1 Scala 2.12

1.1.1 Prezentare

Scala este un limbaj de programare multi-paradigmă ce permite implementarea pattern-urilor de programare uzuale intr-o manieră elegantă, concisă şi type-safe[1]. Acesta îmbină principiile programării orientate pe obiect cu cele ale programării funcționale.

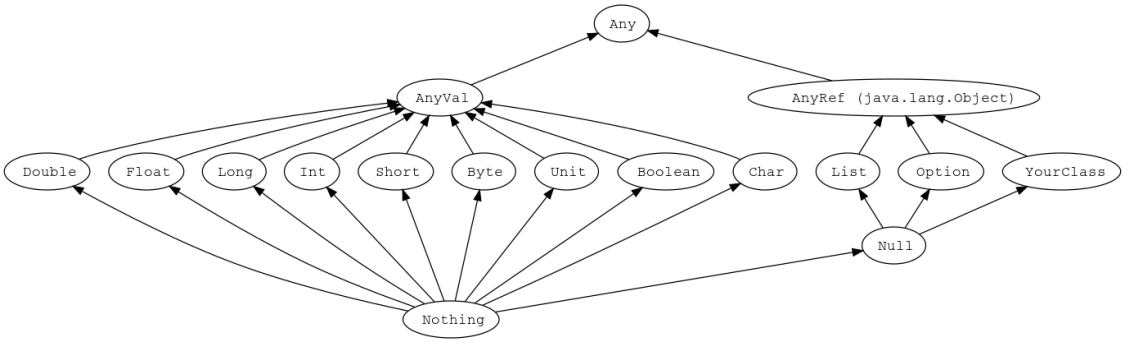

fig 1.1 Ierarhia tipurilor în Scala

Scala se încadrează în familia limbajelor orientate pe obiect datorită faptului că fiecare valoare este un obiect, ierarhia claselor în Scala fiind ilustrată în diagrama următoare:

Tipul și comportamentul unui obiect sunt definite de clase si trait-uri[2]. Clasele pot fi extinse prin derivare, cât și printr-un mecanism de compunere pe bază de “mixins” ce servește drept înlocuitor pentru moștenirea multiplă.

[1] Un limbaj de programare este considerat “type-safe” dacă singurele operații ce pot fi executate asupra datelor sunt cele acceptate de tipul datelor.

[2] Trait-urile sunt folosite pentru a defini seturi de câmpuri și interfețe ce aparțin mai multor clase. Sunt similare interfețelor din Scala8. Clasele și obiectele pot extinde trait-uri, însă acestea nu pot fi instanțiate și prin urmare nu pot avea parametri.

Scala este de asemenea parte din familia limbajelor de programare funcționale datorită faptului că fiecare funcție este o valoare. Scala oferă o sintaxă succintă (lightweight syntax) pentru a defini funcții anonime[3], funcții de tipul high-order functions[4], permite definirea de funcții în interiorul altor funcții - numite și nested functions, și de asemenea permite definirea metodelor cu multiple liste de parametri (procdeu numit currying[5]).

1.1.2 Clase și obiecte. Obiecte companion

Contribuind la caracterul funcțional al limbajului sunt obiectele de tip "singleton". Acestea sunt ideale pentru a grupa funcțiile ce nu aparțin unei clase anume. Obiectele singleton sunt de tip "lazy", adică sunt construite când sunt referențiate prima oară. Atunci când sunt definite în interiorul unei clase, acestea se comportă ca val-urile de tip lazy.

Pentru orice clasă definită, Scala permite definirea unui "companion Object". Pentru o clasă definită *List*, obiectul companion poate fi caraceritizat astfel:

"Adesea, vom declara un *companion object* pe lângă tipul de date definit și constructorii săi. Acesta este doar un *obiect* cu același nume ca tipul de date (în acest caz *List*) une pune numeroase funcții ajutătoare pentru crearea sau manipularea valorilor acestui tip de date.

Dacă, spre exemplu, am dori să definim o funcție `def fill[A](n: Int, a: A): List[A]` care ar crea o listă cu `n` copii ale elementului `a`, obiectul companion `List` ar fi un loc potrivit pentru aceasta. Obiectele companion sunt mai mult o convenție în Scala. Am fi putut numi acest obiect oricum, însă numindu-l `List` am făcut clar faptul că funcțiile acestuia au legătură cu listele." (Chiusano, Bjarnason, 2)

Obiectele companion pot fi privite ca un "repository" pentru funcțiile ce sunt asociate unei clase, însă nu depind de o instanța specifică. Ele sunt potrivite, pe lângă declararea funcțiilor utilitare, pentru a declara funcții ce definesc interacțiunea cu alte module sau tipuri din program. Spre exemplu dacă am avea o funcție ce descompune o instanță în membrii ei constituenți, ar fi mai potrivit să o definim în obiectul companion

[3] Funcțiile anonime sunt funcții ce nu au un identificator, ex `(param: Int) => param + 1`
[4] High-order functions sunt funcții ce primesc ca parametru sau întorc alte funcții, ex: `def promovare(salarii: List[Double], criteriu: Double => Double)`
[5] Ex: `def funcțieCurrying(param1: Int)(param2: Double, param3: String): String`

deoarece logica de extragere a membrilor nu depinde de instanța clasei. Aceste tipuri de funcții se numesc *extractors*. Vom reveni mai târziu asupra lor.

1.1.3 Case classes

"Spre deosebire de clasele obișnuite din Scala, un *case class* generează mare parte din cod în locul tău, cu următoarele avantaje:

- este creată funcția *apply*, astfel încât nu este nevoie să folosești operatorul new pentru a crea o instanță;
- sunt create metode accesor pentru toți parametrii constructorului, deoarece parametrii constructorilor case class-ului sunt publici și imutabili - adică de tip *val*;
- este creată metoda *unapply* ce facilitează folosirea case class-urilor în expresii *match*;
- metodele *equals* și *hashCode* ce permite compararea obiectelor și constuirea cu ușurință a *map-urilor* pe baza lor;" - (Alexander, 1)

Prin intermediul structurii "case class" și a încorporării mecanismului "pattern-matching", Scala oferă funcționalitatea tipurilor algebrice, un element comun în multe limbaje funcționale.

```
case class CaseClassEx(
                          camp1: String,
                          camp2: Int
                        )

CaseClassEx("exemplu", 0) == CaseClassEx("exemplu", 0)  true: Boolean
```

Case class-urile sunt tipuri specializate pentru programarea funcțională. Aceste au stare imutabilă și prin urmare toate elementele pot fi definite ca publice, funcțiile accesor putând fi apelate în operații de transformare precum *map, filter, flatMap* etc. De asemenea sunt ușor comparabile datorită obiectului companion creat automat ce definește funcția *equals*. Prin urmare ele sunt foarte ușor de folosit în *pattern matching*.

Datorită imutabilității și flexibilității lor, *case class-urile* sunt adesea folosite pentru a reprezenta recordurile dintr-o sursă de date, entitățile request-urilor web, răspunsurile întoarse de un serviciu, mesaje în cadrul unei arhitecturi bazată pe *sisteme de actori*.

1.1.4 Pattern matching

Pattern matching-ul este un mecanism ce permite încadrarea unei valori într-un tipar. O potrivire cu un anumit tipar permite separarea valorii în componentele sale. Este o versiune superioară a instrucțiunii switch din limbajul Java și poate substitui cu ușurință o serie de instrucțiuni de tipul "if/else".

Pattern matching-ul se poate aplica atât la nivel de valoare cât și la nivelul claselor cărora instanța evaluată aparține. În cazul case class-urilor, pattern matching-ul poate fi folosit cu ușurință pentru a descompune instanța comparată în parametrii constituenți în cazul unei potriviri.

```
val pm = List("Exemplu", "Pattern", "Matching") match {
  case Nil ⇒ "Listă goală"
  case "Exemplu" :: tail ⇒ tail
  case _: List[String] ⇒ "Altă listă de string-uri"
}
pm  List(Pattern, Matching): java.io.Serializable
```

În exemplul de mai sus, primul caz reprezintă comparația pe bază de tip. În Scala obiectul *Nil* extinde clasa *List* și reprezintă lista goală.

Al doilea caz reprezintă categoria listelor care încep cu elementul *"Exemplu"*. Valoarea elementului *tail* nu este importantă. Operația de pattern matching va evalua întâi dacă elementul conține cel puțin un element, după care va verifica dacă valoarea primului element este egală cu *"Exemplu"*. În cazul unei potriviri, operația va întoarce tail-ul listei, adică lista inițială minus primul element.

Ultimul caz verifică dacă lista este de tip *List[String]* și întoarce un string.

În cazul operației de pattern matching, utilizatorul nu poate doar să definească doar valoarea returnată ci poate defini o logică particulară de execuție pentru fiecare caz.

Spre exemplu, dacă am concepe codul pentru un dispozitiv ce răspunde la comenzi vocal simple, implementarea ar putea arăta astfel:

```
def receive(command: String): Either[String, Task] = command match {
  case "Start" ⇒ Right(StartTask)
  case "Stop" ⇒ Right(StopTask)
  case "Pause" ⇒ Right(PauseTask)
  case _: String ⇒ Left("Command unknown")
}
```

Return type-ul *Either* este un tip de date din Scala folosit adesea pentru a trata cazurile de eroare. Vom reveni asupra modurilor funcționale de tratare a excepțiilor în Scala în următorul subcapitol.

O astfel de funcție este utilă deoarece poate fi legată cu ușurință într-o serie de operații de *map* sau într-un *for comprehension*. Spre exemplu, considerând că `StartTask`, `StopTask` și `PauseTask` extind `Task` am putea defini o funcție `def executeTask(t: Task): Boolean = { … }` ce execută taskul și întoarce dacă acesta a fost executat cu succes.

Cazurile unei operații de pattern matching pot avea pattern guards - adică pe lângă aparținerea la tiparul în cauză, valoarea evaluată trebuie să satisfacă și o condițională. Spre exemplu putem avea:

```
def displayNotification(notification: Notification): String = {
  case email: Email if(email.contains("important")) ⇒ "Important email"
  case email: Email ⇒ "Regular email"
  case missedCall: Call ⇒ "Missed call"
}
```

De observat este că pentru funcțiile cu un singur parametru nu mai este necesară folosirea keywordului *match*. Aceste funcții sunt numite *case functions*.

1.1.5 Exception handling folosind Either, Option și Try

Mecanismul *try catch* nu este compatibil cu paradigma programării funcționale. Un principiu al programării funcționale este că o funcție ar trebui să facă un singur lucru și nu ar trebui să aibă efecte secundare - adică să afecteze starea programului. Prin definiția sa *try catch* este responsabil atât cu executarea codului cât și tratarea excepțiilor ce apar.

Pentru a trata excepțiile în manieră funcțională, Scala pune la dispoziția utilizatorilor trei tipuri: *Either*, *Option* și *Try*.

Either este folosit pentru funcțiile ce întorc două tipuri, tipul pentru cazul de succes și cel pentru cazul de eroare. *Either[A, B]* este clasa de bază pentru clasele derivate *Left[A]* și *Right[B]*. Tipul *Right* este folosit pentru încapsularea valorilor în cazul procesării cu succes, iar *Left* pentru a încapsula valorile întoarse pentru cazul de eșec.

Tipul Option este descris în modul următor:

“ Scala are un tip standard numit *Option* pentru valori opționale. Astfel de valori pot lua două forme. Pot fi de forma *Some(x)* unde *x* reprezintă valoarea, sau poate fi de tip *None* - ce reprezintă absența unei valori.” (Odersky, Spoon, Venners, 3)

Tipul *Option* este adesea returnat de colecțiile standard din Scala. Spre exemplu apelarea metodei *get* pentru un *map* poate întoarce *Some(valoare)*, când cheia dată ca parametru metodei *get* există în *map,* sau *None* în caz contrar.

Spre exemplu:

```
val map = Map('a' -> 1, 'b' -> 2)
```

`map.get('a')` va întoarce `Some(1)`

`map.get('c')` va întoarce `None`

Tipul *Try* este asemănător tipului *Either*. Acesta este derivat de clasele *Success* și *Failure*. Tipul *Success* este asemănător ca scop tipului *Right*, iar tipul *Failure* este de regulă un wrapper peste o excepție.

1.1.6 Type system

Scala beneficiază de un type system[6] care asigură, în timpul compilării, că abstracțiile și tipurile definite sunt folosite în mod corect. Acesta permite folosirea structurilor:

- Clase generice: clasele generice primesc un tip de dată ca parametru; sunt utilizate în special pentru clasele de colecții. Spre exemplu:

```
def grupare[T](elem1: T, elem2: T): List[T] = List(elem1, elem2)
```

- Anotații de varianță: varianța reprezintă corelarea dintre relațiile de derivare dintre tipurile complexe și relațiile de derivare dintre tipurile componentelor acestora.

Pentru un tip generic `A`, putem defini o clasă *covariantă* în `A class` **`Cov[+A]`**, ceea ce definește că pentru un tip `B`, unde `A` este un subtip a lui B, atunci **`Cov[A]`** va fi un subtip a lui **`Cov[B]`**.

De asemenea pentru două tipuri generice `A` și `B`, unde `B` este un subtip al lui `A`, putem defini o clasă *contravariantă* în `A`, spre exemplu **`Con[-A]`**, ceea ce implică că **`Con[B]`** este un subtip al lui **`Con[A]`**.

[6] “Type system” reprezintă un set de reguli care asignează un “tip” tuturor elementelor componente ale unui program, precum variabile, expresii, funcții și module.

În ultimul rând, avem relația de *invarianță*, unde pentru un tip generic `A` putem defini o clasă generică **`Inv[A]`**, ceea ce înseamnă ca nu există nici o ierarhie de tipuri între o instanță **`Inv`** de tipul **`A`** și instanțe ale clasei **`Inv`** de subtipuri sau supertipuri ale lui **`A`**.

- Mecanismul de constrângere a tipurilor folosind "upper type bound" și "lower type bound", ceea ce înseamnă că putem limita parametrii permiși într-un constructor fie la subtipuri ale unui tip generic dat, fie la supertipuri ale acestuia. Pentru a exemplifica mecanismul de "upper type bound" definim o clasă abstractă **`abstract class A`**, o altă clasă abstractă **`abstract class B extends A`** și două implementări concrete **`class C extends A`** și **`class D extends B`**. Atunci pentru o nouă clasă **`class Container[E <: B]`**, instanțierea **`new Container[D](new D)`** ar fi permisă, pe când nu ar fi permisă instanțierea **`new Container[C](new C)`**.
- Clase interioare și tipuri abstracte ca membri ai obiectelor. Spre exemplu pentru un **`trait T`**, putem avea un membru abstract **`Type A`**, ceea ce ar însemna că pentru a implementa `T`, o clasă trebuie sa definească un tip `A`.
- Tipuri compuse, ceea ce permite constrângerea ca parametrii să aparțină unei anumite clase sau să deriveze o anumită clasă de bază și de asemenea să implementeze anumite trait-uri. Spre exemplu **`def ex(obj: Clonable with Extendable): Clonable = {…}`**.
- Mecanismul de **self-types**, ce reprezintă un mod de a declara că un **trait** trebuie să fie compus cu un alt trait pentru a putea fi "mixed-in" de o clasă, deși traitul inițial nu îl extinde în mod direct pe al doilea. Spre exemplu:

```
trait A

trait B {
  this: A ⇒
  def bMet: Unit = println("Trait-ul B trebuie să fie mixed-in cu trait-ul A")
}

class C extends B with A { ... } // clasa C este obligată să integreze A pentru a putea extinde B
```

- Parametri și metode *implicite*, unde:

O metodă în limbajul Scala poate primi o listă de parametri "impliciți". Aceștia sunt notați cu keywordul **implicit**. Dacă parametrii din această listă nu sunt transmiși atunci când funcția este apelată, Scala va verifica dacă poate obține o valoare "implicită" pentru aceștia, care să se potrivească la nivel de tip, și dacă este posibil îi va transmite funcției.

Scala va căuta valorile implicite în două locuri: inițial va verifica definițiile implicite și parametrii impliciți ce pot fi accesați în mod direct în momentul în care funcția este apelată. Dacă nu reușește să satisfacă toți parametrii impliciți, atunci va verifica dacă există valori implicite definite în obiectele companion (**companion objects**)[7] asociate cu candidații tipurilor implicite.

Conversiile implicite constau în faptul că pentrr un o listă de parametri, în locul unui parametru de tip A, se poate trimite un parametru de tip B dacă există definită o funcție implicită `implicit def conversie(a: A): B = { … }` care să fie accesibilă în punctul în care funcția este apelată.

Un exemplu de funcție cu parametri impliciți este:

```
case class ClasăTest(v: String)

implicit val valoareImplicită = ClasăTest("test")

def funcțieCuParamImplciți(implicit param: ClasăTest): String = param.v
funcțieCuParamImplciți  test: String
```

- Metode *polimorfice*; metodele din Scala sunt parametrizabile după tip. Sintaxa este asemănătoare cu cea a claselor generice. Parametri pentru tipuri sunt încapsulați în paranteze pătrate, pe când cei pentru valori sunt încapsulați în paranteze rotunde. Un exemplu de funcție cu parametri pentru tip ar fi:

  `def func[T](par1: T, par2: String) = { … }`.

Scala beneficiază de asemenea de inferența tipurilor, ceea ce înseamnă că utilizatorul nu este nevoit să includă informații suplimentare în definiția valorilor sau funcțiilor. Exemple în care acest mecanism este observabil sunt declararea valorii: **`val valoare = "inferred type"`** și definiția funcției: **`def laPătrat(nr: Int) = nr * nr`**.

În combinație, toate caracteristicile definite mai sus constituie o fundație pentru implementarea tipurilor abstracte reutilizabile și pentru extinderea într-un mod **type-safe** a programelor software.

Nu în ultimul rând, Scala este conceput să fie interoperabil cu **Java Runtime Environment**(JRE). În particular, interacțiunea cu limbajul orientat pe obiect Java este

[7] Companion object-ul unei clase este create prin definirea unui obiect (implicit de tip singleton) ce poartă același nume precum clasa care se dorește a fi acompaniată. Pentru obiectul definit, clasa poartă denumirea de **companion class**. Clasa și obiectul au acces reciproc la toate câmpurile companionului, incluse cele private.

foarte facilă. Scala rulează pe **Java Virtual Machine(JVM)**; codul Scala este compilat în **Java Byte Code** care este apoi executat de către Java Virtual Machine.

Funcționalități noi apărute în Java, precum SAMs(single abstract methods), lambda-funcții[8], annotations[9] și generics au implementări analog în Scala.

Funcționalitățile din Scala ce nu au echivalent în Java, precum parametrii default[10] și *named parameters* sunt compilate cât mai aproape cu putință de codul Java. Scala are același procedeu de compilare ca cel din Java și permite accesul la mii de librării existente pentru Java.

1.2 Apache Kafka

1.2.1 Prezentare

Apache Kafka este definit ca o platformă distribuită de streaming. Bazat pe această definiție putem prezenta caracteristicile platformei explicând conceptele fundamentale.

O platformă de streaming are trei capabilități de bază:

- Să poată **publica** (publish) date în stream-uri de date și să se poată **abona**(subscribe) la un stream de date, funcționând în acest sens similar unei cozi de mesaje sau unui sistem de mesaje de tip enterprise. În alte cuvinte, **publish** reprezintă operația de scriere a datelor, iar **subscribe** reprezintă operația de citire.
- Să poată stoca stream-urile de record-uri într-o manieră **fault tolerant** și **durabilă**. Prin *fault tolerance* se înțelege continuarea funcționalității atunci când o componentă de infrastructură, cum ar fi un nod sau un centru de date, nu își poate continua funcționarea. Prin durabilitate se înțelege că integritatea datelor nu este afectată, că acestea nu sunt corupte de-a lungul timpului. Pe când fault tolerance se referă la redundanța componentelor hardware, în sensul că se alocă, în mod intenționat, un surplus de resurse cu scopul ca în cazul eșecului din partea unei

[8] Lambda funcțiile, numite și Small Anonymous Functions, sunt blocuri funcționale independente ce pot fi transmise în cadrul programului.

[9] Annotations reprezintă o formă de date sintactică ce oferă informații despre alte date din cadrul programului; exemple de annotations sunt `@override`, `@deprecated` etc.

[10] O funcție ce are parametri `default` specifică valori predefinite pentru acei parametri, iar atunci când este apelată iar acei parametri nu sunt specificați le va atribui valorile "default". Exemplu de astfel de funcție este: `def funcțieCuDefault(x: Int, y: Int = 0) = { … }`

componente, celelalte pot prelua sarcinile acesteia și continua procesul neafectate. În cazul durabilității este vorba de redundanța datelor, astfel încât datele sunt replicate ca în cazul pierderii unui set de date, acesta este recuperabil și prin urmare datele nu pot fi compromise sau pierdute.

- Să poată procesa stream-uri de record-uri atunci în momentul apariției acestora.

În general, Kafka este folosit în două tipuri de aplicații. Primul tip este construirea de pipeline-uri pentru streaming de date în timp real, ce sunt capabile să transporte datele între sisteme sau între aplicații într-un mod cât mai fiabil. Al doilea tip de aplicație constă în construirea aplicațiilor bazate pe streaming în timp real, ce transformă datele sau reacționează la datele primite prin intermediul unui stream.

Câteva concepte de bază ce stau la baza platformei Kafka sunt:

- Kafka rulează sub forma unui **cluster** - un set de componente interconectate ce acționează asemenea unui sistem unitar, unde fiecare nod al unui cluster execută aceeași sarcină și este controlat de către software, pe unul sau mai multe servere care pot crea la rândul lor mai multe **datacenters** - un software data center reprezintă un data center ce a fost virtualizat în întregime, adică toată infrastructura a fost virtualizată și percepută ca un serviciu.
- Clusterul de Kafka stochează stream-urile de recorduri în categorii numite **topics**.
- Fiecare record dintr-un topic este format dintr-o cheie, o valoare și un timestamp - timpul la care recordul a fost stocat.

Kafka oferă patru API[11]-uri de bază:

- API-ul **Producer** permite aplicațiilor să publice un stream de record-uri într-unul sau mai multe topicuri de Kafka.
- API-ul **Consumer** permite unei aplicații să se aboneze la unul sau mai multe topicuri și să proceseze streamurile de recorduri pe care acestea le primesc.
- API-ul **Streams** permite unei aplicații să ia rolul unui **stream processor**, fiind astfel capabil să consume un stream de intrare - input stream, de la unul sau mai multe topicuri și să producă un stream de ieșire pe care să îl trimită către unul

[11] Un API reprezintă un set de funcții și proceduri ce permit crearea de aplicații ce accesează funcționalitățile unei aplicații și datele unei aplicații sau a unui serviciu.

sau mai multe topicuri de ieșire - output topics, transformând un stream de date de intrare într-un stream de date de ieșire.

- API-ul **Connector** permite construirea și reutilizarea de producers și consumers și conectează topicurile de Kafka cu aplicații sau sisteme de date existente. Spre exemplu, un conector către o bază de date relațională ar putea avea rolul de a înregistra fiecare schimbare a tabelei din Kafka.

În cadrul Kafka, comunicarea dintre servere și clienți este realizată prin intermediul unui TCP(Transmission Control protocol) performant.

1.2.2 Topics

Topicul reprezintă structura de abstracție fundamentală pe care Kafka o oferă pentru un stream de recorduri.

Un topic reprezintă o categorie sau "feed" în care recordurile sunt publicate. Topicurile de Kafka sunt de tip *multi-subscriber* - pot avea zero, unul sau mai mulți consumeri abonați la datele publicate în acesta.

Pentru fiecare topic, clusterul de Kafka păstrează un log partiționat cu o structură de forma:

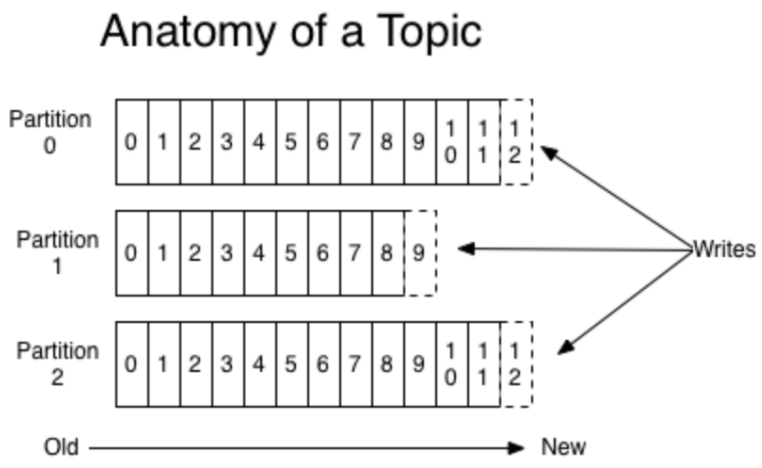

fig 1.2 Anatomia unui topic Kafka

Fiecare partiție este o secvență de recorduri ordonată și imutabilă, căreia îi sunt continuu anexate noi date; arhitectura poate fi definită ca **structured commit log**. Fiecărui

record din cadrul partițiilor îi este asignat un număr de identificare secvențial, numit și **offset**, ce este folosit pentru a identifica fiecare record din cadrul partiției.

Kafka are performanță constantă raportat la dimensiunea datelor stocate, ceea ce face ca stocarea datelor pe termen lung sa nu producă niciun impediment.

"Singura formă de metadata păstrată la nivel de consumer este offsetul la care acesta a ajuns în cadrul logului. Consumerul are control asupra offsetului de la care citește. Deși în mod obișnuit un consumer citește în mod secvențial dintr-un topic, offest-ul crescând linear, consumerul are mereu posibilitatea de a schimba poziția la care se află, putând astfel consuma date în orice ordine dorește" (apache kafka quickstart, 7).

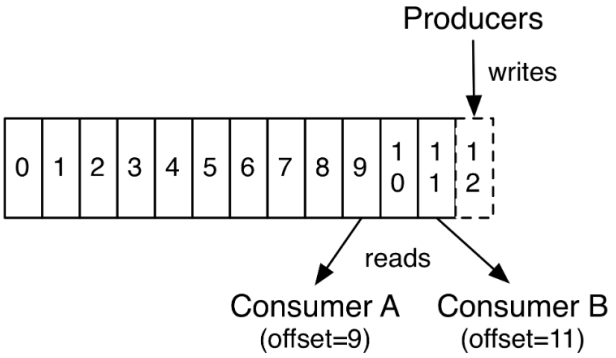

fig 1.3 Citirea de la offseturi configurabile

Acest mix de mecanisme asigură eficacitatea și costul redus a consumerilor, crearea și stoparea, cât și activitatea lor neavând un impact semnificativ asupra clusterului sau asupra celorlalți consumeri.

Partiționarea permite scalarea logului la dimensiuni mai mari decât spațiul disponibil pe un singur server. Fiecare partiție trebuie să se încadreze din punct de vedere al dimensiunii pe serverul host, însă un topic poate avea partiții multiple, stocate pe servere multiple, ceea ce îi conferă capacitatea de a menține cantități arbitrare de date. Un alt rol al partițiilor este cel de unități pentru paralelism.

Partițiile unui log sunt distribuite între servere în clusterul de Kafka, fiecare server procesând o anumită cantitate de date și request-uri pentru un segment din partiții. Partițiile sunt **replicate** într-un număr configurabil de servere pentru a oferi **fault tolerance**.

1.2.3. Brokers și Clusters

“Un singur server de Kafka este numit un `broker`. Brokerul primește mesaje de la producers, le atribuie un *offset* și apoi îi scrie în disk storage. Acesta este responsabil și cu servirea consumerilor, răspunzând cererilor adresate către partiții cu șirul de mesaje ce a fost scris pe disc ce corespunde partiției în cauză. Depinzând de calitatea hardware-ului folosit, un broker poate opera cu zeci de mii de partiții și milioane de mesaje.” (Narkhede, Shapira, Palino, 7).

Kafka brokers au fost concepuți pentru a rule în clustere. Pentru fiecare partiție, un server are rolul de **leader**. Acesta poate avea zero sau mai multe servere de tip **follower**. Leaderul este responsabil cu procesarea request-urilor de citire și scriere pentru partiția respectivă, iar followerii replică logica executată de leader. În circumstanța încetării funcționării liderului, unul din serverele follower va deveni noul lider. Un server este lider pentru o anumită partiție și follower pentru un număr configurabil de alte partiții.

Clusterul de Kafka păstrează în mod durabil toate recordurile publicate, indiferent dacă acestea au fost sau nu consumate, folosindu-se de un timp de păstrare (**retention period**) configurabil. Dacă perioada de retention este configurată la o săptămână, atunci recordurile vor fi păstrate timp de o săptămână de la data publicării, timp în care acestea vor putea fi consumate, după care vor fi eliminate pentru a elibera spațiul de memorie.

1.2.4 Producers și Consumers

Producerii publică datele într-un topic la alegerea lor. Aceștia sunt responsabili cu alegerea partiției în care fiecare record este publicat. Aceasta poate fi aleasă fie într-o manieră clasică, de tipul round-robin spre exemplu, fie printr-o funcției de partiționare definită de către producer.

Consumer-ii sunt separați în Consumer Group-uri. Atunci când un record este publicat într-o partiție de Kafka, acesta este trimis către o instanță de consumer din cadrul fiecărui grup de consumeri abonat la partiția respectivă. În cazul în care toți consumerii fac parte din același grup, recordurile vor fi distribuite uniform către consumeri, iar în cazul în care fiecare consumer aparține unui consumer group separat, fiecare record va fi trimis către fiecare grup, într-o manieră broadcast.

Kafka implementează consumarea recordurilor prin împărțirea partițiilor din cadrul log-ului în așa fel încât fiecare instanță de consumer să primească o porțiune aproximativ

egală din partiții. Acest proces de “load-balancing” este efectuat de către protocolul de Kafka în manieră dinamică. Dacă noi instanțe consumer se alătură grupului, atunci acestea vor primi o parte din partițiile fiecărei instanțe deja existente, iar dacă o instanță este oprită, atunci celelalte instanțe vor împărți load-ul care îi revenea acesteia.

Kafka păstrează ordinea recordurilor doar la nivel de partiție, ceea ce înseamnă că recorduri stocate în partiții diferite nu vor avea nici o relație de ordine între ele. Însă ordonarea per partiție în combinație cu posibilitatea de a partiționa în funcție de cheie este suficientă pentru mare parte din aplicațiile client. Dacă se dorește, ordonarea se poate face la nivel de topic dacă și numai dacă acesta conține o singură partiție.

1.2.5 Funcționalitățile Kafka

1.2.5.1 Kafka ca Sistem de Stocare

Orice sistem de cozi de mesaje ce permite separarea logicii de publicare de mesaje de cea de consumare a mesajelor poate fi considerată ca fiind un sistem de stocare pentru mesajele în curs de transmitere. Din această perspectivă, Kafka este considerat un sistem foarte eficace de stocare a mesajelor.

Datele trimise către Kafka sunt salvate pe disc and replicate pentru a asigura fault tolerance. De asemenea produceri-lor li se oferă posibilitate să aștepte confirmarea ca datele au fost primite și replicate de către Kafka și prin urmare vor persiste chiar și în cazul în care serverul către care au fost trimise inițial cedează. Operația de scriere nu este considerată ca fiind completă până când această validare nu este trimisă.

Structurile de disc folosite de Kafka sunt scalabile, Kafka având aceeași performanță fie că pe server persistă o cantitate de date de 10KB sau de 10TB.

Datorită flexibilității oferite clienților în controlarea capului de citire, Kafka poate fi considerat un sistem de fișiere distribuit dedicat performanței ridicat, timpului scăzut de întârziere în operațiile aplicate pe date, replicării și propagării acestora.

1.2.5.2 Kafka pentru Procesare de Stream-uri

Pe lângă oferirea posibilităților de citire, scriere și stocare a datelor, scopul platformei Kafka este de a facilita procesarea în timp real a stream-urilor.

Orice serviciu ce preia un stream de date dintr-un set de topic-uri de tip input, execută un algoritm de procesare asupra acestor date și apoi produce un stream de date de ieșire ce este stocat în topicuri de tip output poate fi considerat un procesor de stream-uri. Spre exemplu un serviciu de livrări poate primi continuu date referitoare la comenzi noi primite, comenzi finalizate și puncte de tranziție în care se află comenzile în desfășurare, putând astfel produce un stream de comenzi și traseul actualizat pe care acestea trebuie să îl parcurgă până la destinație.

API-urile de Consumer și Producer permit o procesare minimală a stream-urile. Pentru operații de procesare complexe Kafka pune la dispoziție **API-ul Streams**, ce îmbină beneficiile și simplitatea dezvoltării aplicațiilor Java și Scala în contextul clientului cu tehnologia bazată pe cluster din Kafka pe partea de server. Acest API oferă programelor posibilitatea de a executa operații non-triviale precum agregări sau join-uri asupra stream-urilor folosind Kafka.

API-ul Streams este construit pe baza primitivelor din Kafka: API-urile de consumer și producer sunt folosite pentru operațiile de intrare-ieșire; Kafka ocupă rolul de stare, de stocare a datelor - **stateful storage**, iar mecanismul de grupare este folosit pentru a asigura fault tolerance în cadrul instanțelor de procesor.

1.2.6 Serializers

Pentru a putea publica recorduri într-un topic, un producer trebuie să fie capabil să transforme un obiect de tipul dorit într-un șir de biți. Din acest motiv, fiecare producer trebuie să specifice în configurația sa un obiect care să conțină logica de transcriere în bytes, numit *Serializer.*

Implicit, Kafka oferă trei tipuri de *Serializers*: pentru *stringuri*, pentru *int* și pentru *byteArray*. Însă aceste tipuri nu sunt suficiente, majoritatea aplicațiilor ce publică recorduri în Kafka dorind să publice tipuri mai complexe decât cele implicite.

Utilizatorii au mai multe opțiuni când doresc definirea unui *Serializer* personalizat. Aceştia pot fie folosi o librărie generică de serializare, cum ar fi Apache Avro, pentru a crea recordurile, fie pot defini ei înșiși un serializator. Este recomandat ca utilizatorii să folosească o librărie generică, întrucât nivelul de abstracție este mai ridicat, optimizat și poate fi scris concis.

Pentru a serializa un obiect folosind Avro, este necesar să definim *schema* recordului. Aceasta are următoarea formă:

```
{
"namespace": "customerManagement.avro",
"type": "record",
"name": "Customer",
      "fields": [
            {"name": "id", "type": "int"},
            {"name": "name", "type": "string"},
            {"name": "email", "type": ["null", "string"],
            "default": "null"}
      ]
} (Narkhede, Shapira, Palino, 5)
```

Schema Avro reprezintă structura recordurilor din topic. O funcție generică de serializare poate fi scrisă astfel:

```
private def encodeToAvro(record: Record, schema: Schema): Array[Byte] = {

  val writer = writers(schema)
  writer.write(record, encoder)
  encoder.flush()
  out.close()
  val byteArray = out.toByteArray
  out.reset()

  byteArray
}
```

1.3 Apache Spark

1.3.1 Prezentare

“Apache Spark este un sistem de calcul computațional în cluster (**cluster computing**). Principala atracție a acestui sistem este viteza foarte mare de procesare, în momentul lansării primei versiuni stabile (2014) acesta permitea rularea de 100 de ori mai

rapidă în memorie și de 10 ori mai rapidă pe disc a programelor decât cea mai populară soluție din perioada respectivă, Hadoop" (Miller, 8).

Spark oferă posibilitatea de a scrie cod rapid, eficient și concis. Pe când un simplu program "Hello World" are o implementare de aproximativ 50 de linii de cod în Java pentru MapReduce[12], echivalentul pentru Spark (scris în Scala) este foarte succint:

```
sparkContext.textFile("hdfs://...")
            .flatMap(line => line.split(" "))
            .map(word => (word, 1)).reduceByKey(_ + _)
            .saveAsTextFile("hdfs://...")
```

Câteva caracteristici ale framework-ului Spark ce îl fac un candidat puternic când vine vorba alegerea unui sistem computațional pentru procesare de set-uri largi de date sunt: numeroasele API-uri furnizate de acesta pentru limbaje precum Java, Scala, Python; posibilitatea de a fi integrat cu ecosistemul Hadoop și sursele de date aferente dintre care amintim HDFS, AmazonS3, Hive, Cassandra și capacitatea de a rula pe clustere administrate de Hadoop YARN sau Apache Mesos.

Funcționalitățile de bază din Spark (numit și **Spark Core**) sunt suplimentate cu un număr puternic de librării precum SparkSQL, Spark Streaming, MLlib - folosită pentru machine learning, precum și alte librării mai puțin populare.

1.3.2 Spark Core și Resilient Data Sets (RDDs)

Spark Core este engine-ul de bază pentru procesarea de date la scară largă, în paralel și în manieră distribuită[13]. Acesta este responsabil cu administrarea memoriei and recuperarea în caz de eșec (fault recovery); cu programarea executării, distribuirea și monitorizarea "job-urilor" în cluster, cât și cu interacțiunea cu sisteme de stocare externe. Spark introduce conceptul de **Resilient Distributed Datasets (RDDs)**.

RDD-urile sunt modelul principal de manevrare a datelor în Spark. Acestea reprezintă colecții distribuite de elemente. În Spark toate funcționalitățile se bazează fie pe

[12] Hadoop MapReduce este un framework software ce are ca scop facilitarea scrierii aplicațiilor dedicate procesării cantităților foarte mari de date în paralel, folosind tehnologia cluster și într-o manieră fault tolerant.
[13] Procesarea distribuită a datelor este o metodă de containerelor virtuale astfel încât acestea formează un sistem de procesare a datelor.

crearea de RDD-uri, transformarea RDD-urilor preexistente sau apelarea executarea de operații asupra acestora în scopul obținerii unui rezultat. Intern, Spark distribuie datele încapsulate de RDD-uri și paralelizează operațiile executate asupra acestora.

RDD-urile sunt create fie pornind de la un fișier din Hadoop file system, sau de la o colecție din Scala deja existentă în programul driver, pe care se aplică apoi operații de transformare. Utilizatorii pot de asemenea menține un RDD în memorie (caching) , permițând astfel ca acesta să fie folosit într-o manieră eficientă în cadrul operațiilor paralele. De asemenea , RDD-urile sunt capabile să se redreseze în cazul în care apar probleme la nivel de nod.

RDD-urile sunt asemănătoare cu colecțiile imutabile, atât secvențiale și cât și paralele din Scala și oferă mare parte din funcțiile oferite de aceste colecții, precum `map, flatMap, filter, reduce, fold, aggregate etc.` Toate funcțiile listate anterior au aceeași funcționalitate atât în Spark cât și în Scala, mai puțin aggregate. Pe când în Scala elementul unitate - "0", este call by-name[14], Spark fiind conceput pentru procesarea în paralel a seturilor mari de date nu poate implementa această funcționalitate deoarece evaluarea elementului pentru fiecare subset de date ar fi prea costisitoare.

Spre deosebire de colecțiile din Scala, RDD-urile nu pot implementa variațiile funcției `reduce`, anume `reduceLeft` și `reduceRight`, deoarece acestea se bazează pe ordinea elementelor în colecție, aspect incompatibil cu procesarea în paralel.

RDD-urile acceptă două tipuri de operații:

- **Transformări** - operații executate asupra unui RDD ce rezultă într-un nou RDD ce conține rezultatele. Operații de acest tip sunt `map, filter, join, union` etc.
- **Acțiuni** - operații ce întorc o valoare după ce execută un șir de computații asupra unui RDD. Operații de acest tip sunt `reduce, count, first` etc.

Transformările în Spark sunt de tip "lazy", adică rezultatul acestora nu este calculat imediat după ce au fost apelate. În schimb, Spark "memorează" operația ce trebuie executată și RDD-ul pe care aceasta trebuie executată. Transformările sunt computate doar în momentul în care o operație de tip **acțiune** este executată asupra RDD-ului, iar rezultatul este returnat către programul driver. Acest design permite framework-ului Spark să ruleze mai eficient. Spre exemplu, dacă în cadrul unui program driver pentru Spark o

[14] În Scala, un parametru call by-name este un parametru evaluat abia în momentul în care acesta este folosit; spre exemplu dacă o funcție primește un parametru by-name, însă acesta nu este folosit nicăieri în corpul funcției, acesta nu va fi evaluat.

serie de transformări ar fi aplicate asupra conținutului unui fișier, urmând ca apoi să fie apelată acțiunea `first`, spark ar aplica transformările doar pentru prima linie în loc de a le aplica pentru întreg fișierul.

Deși procedura default este de a re-aplica transformările asupra unui RDD de fiecare dată când o operație de tip acțiune este efectuată asupra acestuia, prin păstrarea RDD-ului în memorie, folosind metodele `persist` și `cache`, Spark va păstra elementele în cluster pentru a putea fi accesate mai rapid în query-urile următoare.

1.3.3 Programarea cu Tuples Chei-Valoare

Perechile de tip cheie/valoare sunt o structură folosită frecvent de operațiile Spark. RDD-urile de tip cheie-valoare sunt adesea folosite pentru a executa agregări și vor fi adesea responsabile pentru executarea unei logici de tipul **ETL**(extract, transform, load) pentru a converti datele în formatul cheie/valoare. RDD-urile definesc operații pentru lucrul cu perechi cheie/valoare precum gruparea după cheia, uniunea după cheie și altele.

"Există mai multe moduri de a obține RDD-uri de perechi în Spark. Multe din metodele de creare a RDD-urilor prezentate anterior întorc deja RDD-uri de acest tip. Alte dăți dorim obținerea unui RDD de perechi dintr-un RDD obișnuit. În cazul aceste putem aplica operația *map* pe RDD-ul pe care dorim să îl transformăm. Pentru a ilustra acest procedeu vom prezenta în cod cum pornind de la un RDD de linii de text și selectează pe post de cheie primul cuvânt din fiecare linie de text

```
val pairs = lines.map(x => (x.split(" ")(0), x))
```"

(Karau, Konwinski, Wendell, Zaharia, 4)

Pair RDD-urile permit folosirea tuturor operațiilor de transformare existente pentru RDD-uril obișnuite. Singura diferență este că deoarece RDD-urile conțin tupluri de valori, funcțiile date ca parametru trebui să aibă la rândul lor parametri de tip tuplu.

1.3.4 Spark SQL

SparkSQL este o componentă Spark concepută pentru procesarea datelor în manieră structurată. Această componentă permite executarea de query-uri asupra datelor folosind SQL sau Hive Query Language.

Spre deosebire de API-ul RDD-urilor , interfețele oferite de Spark SQL îi oferă Spark-ului mai multe informații despre structura datelor și a operațiilor ce sunt executate.

Există mai multe moduri de a interacționa cu Spark SQL, printre care **SQL** și API-ul **Dataset**. Când un rezultat este computat se folosește același motor de execuție, indiferent de API-ul sau limbajul folosit pentru a exprima calculul. Această unificare le permite utilizatorilor să folosească API-ul cel mai potrivit pentru a exprima transformarea dorită.

Cum am menționat anterior, o funcționalitate a Spark SQL este de a executa SQL queries. Când query-urile sunt rulate din cadrul altui limbaj de programare rezultatele vor fi întoarse ca un **Dataset** sau un **DataFrame**.

Un **Dataset** este o colecție distribuită de date. Aceasta îmbină beneficiile RDD-urilor (strong typing, capacitatea de a folosi funcții lambda) cu optimizările motorului de execuție din Spark SQL. Un Dataset poate fi construit pornind de la obiecte din JVM[15] și apoi poate fi manipulat aplicând transformări funcționale, precum `map`, `flatMap`, `filter` etc.

Un **DataFrame** este un Dataset organizat în coloane nominale. Conceptul, un DataFrame este echivalent cu o tabelă dintr-o bază de date relaționale, fiind optimizată pentru operațiile pe date structurate. Acestea pot fi construite din surse precum: fișiere cu date structurate, tabele din Hive, baze de date externe, RDD-uri.

Spre exemplu avem funcția `groupByKey` ce grupează perechile din RDD în funcție de valoarea cheii acestora. Implementarea acesteia este de forma

`x: List[T, U] => x.groupBy( element => element._1)`, unde funcția parametru din `groupBy` este o funcție anonimă ce pentru o pereche primită ca parametru întoarce primul element al acesteia, adică `_1`.

Un alt exemplu este funcția `reduceByKey(func)`, ce are ca scop combinarea valorilor ce au aceeași cheie. Spre exemplu, dacă am dori calcularea sumei tuturor valorilor ce au aceeași cheie am apela funcția `reduceByKey` cu următoare funcție dată ca parametru: `reduceByKey((x, y) => x + y)`, care pentru un RDD ipotetic ce conține perechile `('a', 1), ('b', 1), ('c', 1), ('a', 1), ('b', 2)` ar întoarce un RDD cu valorile `('a', 2), ('b',3), ('c', 1)`.

[15] Acronim pentru Java Virtual Machine

1.3.5 Spark Streaming

Spark Streaming este o extensie a API-ului Spark Core ce permite procesarea în timp real a datelor transmise prin intermediul stream-urilor, precum log-uri primite de la servere web, social media și numeroase platforme de streaming precum Kafka.

Spark Streaming este o platformă de procesare de stream-uri scalabilă și fault tolerant, ce permite operarea asupra datelor atât în manieră streaming cât și în manieră batch. Datele procesate pot fi transmise către sisteme de fișiere, baze de date, cozi de mesaje și alte sisteme de stocare.

Structura de abstractizare de bază a Spark Streaming este **Discretized Stream**, sau **DStream**, ce reprezintă un stream de date separat în batch-uri multiple de dimensiune redusă. DStream-urile sunt bazate pe RDD-uri, aspect ce permite componentei de Spark Streaming să fie integrată cu ușurință cu alte componente precum MLlib sau Spark SQL.

Motorul de execuție singular și modelul programmatic[16] unificat pentru stream-uri și batch-uri aduc câteva beneficii față sistemele clasice de streaming precum: redresare rapidă în caz de eșec, o mai bună distribuire a sarcinilor (load balancing) și utilizare a resurselor, interogări native folosind librării avansate de procesare precum SQL, machine learning, procesare pe bază de grafuri.

Spark Streaming poate citi din diferite surse de date. Sursele de bază sunt streamurile de fișiere și akka actor stream. Printre sursele adiționale se numără și Kafka. Conectarea la Kafka este prezentată în paragraful următor:

“Apache Kafka este o sursă populară de date datorită vitezei acesteia și a capacității de a face față erorilor. Folosind suportul nativ pentru Kafka putem procesa cu ușurință mesaje dintr-o varietate de topicuri. Pentru a-l folosi este nevoie să includem artefactul Maven `spark-streaming-kafka_2.10` în proiectul nostru. Acesta ne va pune la dispoziție obiectul **KafkaUtils** ce poate folosi StreamingContext sau JavaStreamingContext pentru a crea un DStream de mesaje Kafka.[...] Pentru a crea un stream vom apela metoda *CreateStream()* folosind streaming contextul creat, un string separat prin virgulă ce va conține hosturile ZooKeeper, numele consumer grupului nostru

[16] Un model programatic reprezintă o abstractizare a sistemului computațional ce permite definirea algoritmilor și structurilor de date

și un map de la topicuri la număr threadurilor ce se folosi pentru a citi din acestea.” (Karau, Konwinski, Wendell, Zaharia, 4)

Partea 2

Considerații privind dezvoltarea aplicației

2.1 Componența aplicației

2.1.1 Prezentare

Aplicația este compusă din două module: **Traffic Generator** și **Traffic Analyzer**. Ambele module sunt scrise în limbajul Scala și folosesc Kafka pentru citirea și scrierea datelor. Traffic Analyzer folosește Spark pentru a procesa datele.

Modulul **Traffic Generator** are rolul de a genera date potrivite structural, în cantități suficiente, pe care modulul **Traffic Analyzer** să poată executa operații analitice optimizate pentru **big data**[17]. Acest modul este implementat folosind API-ul Open Street Routes Map(OSRM). OSRM furnizează generatorului de trafic traseul cel mai scurt dintre două coordonate GPS, definite prin latitudine și longitudine, sub forma unui șir de puncte GPS intermediare, situate în elemente de traseu precum intersecții, sensuri giratorii, puncte de cotitură. Punctele intermediare sunt însoțite de distanța și timpul estimat până la următorul punct intermediar din șir, pe baza cărora este calculată viteza medie estimată a autovehiculelor pe porțiunea respectivă de drum.

Modulul Traffic Generator creează un număr configurabil de autovehicule simulate, iar pentru acestea generează date GPS în timp real, poziția la care se află o unitate fiind înregistrată și publicată într-un topic de **Kafka** - **completed steps**, la un interval mediu de aproximativ o secundă. Deplasarea unui autovehicul simulat este calculată folosind viteza

[17] Big data este un termen folosit pentru a descrie date cu dimensiuni foarte mari sau în cantități foarte mari, ce se dorește a fi prelucrate cât mai eficient și stocate cât mai ieftin din punct de vedere economic

medie obținută din răspunsul API-ul OSRM și diferența de timp față de pasul înregistrat anterior.

Modulul de **analiză al traficului** se bazează pe **Kafka** pentru obținerea datelor, iar pentru procesarea lor folosește **Spark**, predominant componenta de **Spark Streaming**.

Traffic Analyzer este compus din două servicii: **Traffic Alerts** și **Traffic Density**.

Serviciul **Traffic Alerts** constă în monitorizarea fluxului de trafic în timp real și semnalarea tuturor zonelor, de rază configurabilă, unde numărul de mașini depășește un prag - de asemenea configurabil. Datele sunt consumate în **manieră streaming**, sunt procesate folosind DStream-uri și RDD-uri, apoi tot în manieră streaming sunt publicate în topicul de **traffic-alerts**. Din acest topic alertele de trafic pot fi consumate de un alt consumer și transmise către o aplicație client, ce dorește calcularea celor mai eficiente rute în trafic în timp real, evitarea zonelor aglomerate, asemeni aplicațiilor Waze sau Bolt.

Serviciul **Traffic Density** operează pe datele dintr-o perioadă anterioară, configurabilă și crează pe baza acestora și a unei arii configurabile o listă de subarii ale ariei date, a căror suprafață este definită de către utilizator, ordonate în funcției de densitate traficului în intervalul de timp cerut, unde densitatea este definită ca suma numărului de pași făcuți de fiecare autovehicul în cadrul unei zone.

Pașii sunt citiți în **manieră batch** din topicul **completed-steps** și sunt procesați folosind doar RDD-uri. Lista subariilor acompaniate de densitățile înregistrate pe parcursul perioadei date pentru acestea sunt scrise în manieră batch în topicul de Kafka - **area-density**.

2.1.2 OSRM API

Pentru o mai bună înțelegere a structurii datelor generate, vom face o scurtă prezentare a API-ului OSRM și a resurselor puse la dispoziție de acesta.

Traffic Generator folosește serviciul `Route` oferit de OSRM pentru a genera traseele parcurse de unitățile de trafic. Pentru o listă de coordonate geografice, acesta întoarce întoarce un traseu - definit ca o listă de pași reper pentru a ajunge din primul punct din lista de coordonate până în ultimul punct al acesteia.

Formatul request-urilor acceptate de serviciul route este următorul:

`/route/v1/`**`{profile}`**`/`**`{coordinates}`**`?alternatives={true|false}&steps={true|false}&geometries={polyline|polyline6|geojson}&o`

`verview={full|simplified|false}&annotations={true|false}`, unde parametrii requestului au următoarele semnificații:

- `alternatives:` specifică dacă apelantul dorește ca OSRM să includă în răspuns și rute alternative, pe lângă ruta principală calculată;
- `steps:` specifică dacă apelantul dorește includerea pașilor pentru subsecțiuni ale rutei;
- `geometries:` specifică dacă utilizator dorește informații structurale despre segmentele traseului;
- `overview:` specifică tipul de perspectivă asupra "geometriei" traseului; aceasta poate fi completă, simplificată sau absentă; `overview` poate fi perceput ca întorcând reprezentarea grafică a traseului;
- `annotations:` specifică dacă se dorește ca elementul `geometry` să fie însoțit de notații.

Dintre toate aceste informații Traffic Generator are nevoie doar de lista pașilor.

2.2 Traffic Generator

Modulul Traffic Generator servește ca sursă de date pentru modulele analitice. Acesta simulează traficul rutier în **timp real**, pe o suprafață definită de utilizator. Datele generate sunt scrise într-un topic de Kafka - **completed steps**, în manieră streaming.

2.2.1 Comunicarea cu utilizatorul

Modulul Traffic Generator pune la dispoziția utilizatorului un endpoint de tip POS[18], numit **generateTraffic**. Requestul POST trebuie să conțină în body o entitate de tip JSON sau parsabilă în JSON, care să aibă următoarele câmpuri:

- **center**: este un sub-obiect compus la rândul său din două câmpuri: **latitude** și **longitude**. Acest câmp reprezintă centrul ariei geografice în care se dorește simularea traficului. Formatul este cel al unei coordonate geografice - compusă din latitudine și longitudine. Aria este definită ca un cerc de rază **r** configurabilă, de origine **center**.

[18] POST este o metodă a protocolului HTTP folosită pentru a trimite o entitate către o sursă specifică, adeseori rezultând într-o schimbare de stare.

- **radius**: reprezintă raza ariei din jurul centrului specificat prin parametrul anterior. Unitatea de măsură în care este exprimată este **km**.
- **numberOfUnits**: reprezintă numărul de unități ce se dorește a fi create. Pentru fiecare unitate va fi simulată activitatea în trafic.

În cazul în care entitatea body a requestului nu respectă acest format, aplicația va genera un mesaj de eroare de tipul : "`Malformed request content`". Spre exemplu, dacă cererea nu conține câmpul *numberOfUnits*, atunci mesajul de eroare ar fi însoțit de mesajul: "`Object is missing required member 'numberOfUnits'` ".

Un exemplu de request entity este următorul:

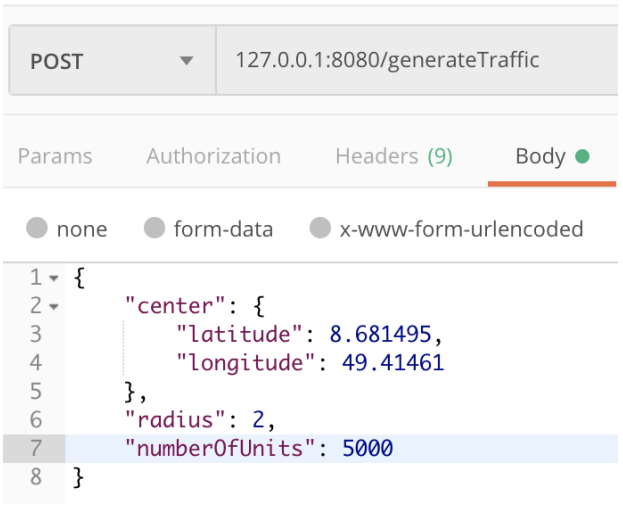

fig 2.1 Exemplu de request pentru serviciul Traffic Generator

2.2.2 Comunicarea cu serviciul Route

Funcția din cod ce se ocupă cu obținerea informațiilor de la serviciul `Route` este următoarea:

```
def getOSMRoute(coordinates: (Coordinate, Coordinate)): Future[OSMResponse] = {
  getHTTPResponse(coordinates).flatMap(toOSMRoute)
}
```

Funcția `getHTTPResponse` se ocupă cu apelarea endpointului și obținerea răspunsului, iar funcția `toOSMRRoute` se ocupă cu convertirea acestuia în tipul de date folosit de program.

Implementarea `getHTTPResponse` se folosește de Akka HTTP pentru a trimite requestul și a aștepta răspunsul în manieră concurentă.

```
def getHTTPResponse(coordinates: (Coordinate, Coordinate)): Future[HttpResponse] = {
  http.singleRequest(HttpRequest(GET, getRequestURI(coordinates)))
}
```

Funcția `toHTTPResponse` folosește un custom *unmarshaller* pentru a despacheta obiectul din răspuns și este responsabilă pentru tratarea cazurilor în care, din varii motive, nu este întors un răspuns cu succes.

```
def toOSMRoute(httpResponse: HttpResponse): Future[OSMResponse] = httpResponse match {
  case HttpResponse(StatusCodes.OK, _, entity, _) =>
    Unmarshal(entity).to[OSMResponse]
  case HttpResponse(status, _, entity, _) =>
    Future.failed(new Exception(handleEndpointError(entity, status)))
}
```

2.2.3 Akka HTTP

Modulul oferă acest API de tip REST[19] prin intermediul librăriei **Akka HTTP**. Akka HTTP este definit ca fiind un set de instrumente pentru a implementa servicii HTTP de tip **provider** sau **consumer**. Aceasta este bazată pe librăriile **akka-actor** și **akka-stream**, akka-actor urmând să fie prezentat mai în detaliu în capitolele următoare. Modulele Akka HTTP pot fi folosite pentru a implementa atât logica de client HTTP cât și logica de server.

Dintre aceste API-uri, Traffic Generator folosește **Core Server API** și **Route DSL**. **API-ul Core Server** este responsabil cu funcționalitățile esențiale ale unui server HTTP, anume: managementul conexiunilor, parsarea și randarea headerelor și a mesajelor, supervizarea logicii timeout pentru request-uri și conexiuni, ordonarea răspunsurilor. Serverul de Akka HTTP este implementat pe baza tehnologiei de streaming din Akka Streams.

Routing DSL este o adiție la API-ul Spark Core ce permite implementarea elegantă a serviciilor web de tip **RESTful**. Routing DSL oferă funcționalitățile complexe specifice serverelor și framework-urilor web, precum deconstrucția URI-urilor, evaluarea conținutului request-urilor și furnizarea de entități ca răspuns (static content serving).

[19] Representational State Transfer - abreviat REST, este o arhitectură software ce permite interoperabilitate dintre un computer și o resursă web printr-un set predefinit de operații fără efecte secundare, adică operații ce nu modifică starea serviciului.

```
val supervisorActor: ActorRef

implicit lazy val timeout: Timeout = Timeout(10.seconds)

lazy val routes: Route =
  pathPrefix( pm = "generateTraffic") {
    pathEnd {
      post {
        entity(as[GenerateRequest]) { request =>
          supervisorActor ! request
          complete(HttpResponse(status = StatusCodes.OK, entity = "Request sent"))
        }
      }
    }
  }
```

Endpoint-ul **generate traffic** este implementat folosind Route DSL. Logica acestuia este foarte simplă: expune endpoint-ul de tip post, parsează entitatea din request ca o instanță a **case class**-ului `GenerateRequest`, trimite instanța `GenerateRequest` către **sistemul de actori** pentru a fi procesat și trimite un mesaj de confirmare către sursa request-ului ce semnalizează că requestul a fost primit.

În cod, metoda **`pathPrefix`** definește numele endpoint-ului, metoda **`post`** specifică că verbul HTTP prin care această resursă este accesibilă este verbul POST. Metoda **`entity`** este responsabilă cu "despachetarea" (procedeu numit unmarshalling) entității primite prin request, iar funcția **`as[T]`** conține logica pentru transformarea unui `byteArray` în **`T`**, în cazul curent tipul parametru fiind **`GenerateRequest`**.

Metoda **`complete`** este responsabilă pentru crearea unui răspuns pentru sursa request-ului. În cazul de față, pentru un request corect formatat se va întoarce un răspuns HTTP cu statusul OK și mesajul, de tip string, "Request sent".

Un server HTTP este instanțiat folosind această rută:

```
serverBinding.onComplete {
  case Success(bound) =>
    println(s"Server online at http://${bound.localAddress.getHostString}:${bound.localAddress.getPort}/")
  case Failure(e) =>
    Console.err.println(s"Server could not start!")
    e.printStackTrace()
    system.terminate()
}
```

2.2.4 Akka actors și Sistemul de actori

Pentru a putea simula activitatea în trafic a unui număr ridicat de autovehicule în paralel, Traffic Generator implementează design pattern-ul[20] **actor system**, provenit din Akka actor.

Un **Actor Model** reprezintă un model matematic de computare în manieră concurentă, unde *actorul* reprezintă unitatea de bază universală a computației concurente. Actorii comunică cu restul aplicației, cât și cu alți actori prin intermediul **mesajelor**. În funcție de tipul mesajului primit un actor este capabil să ia decizii logice, să trimită mai multe mesaje și să determine cum să reacționeze la următoarele mesaje primite. Actorii sunt capabili să modifice starea lor internă, însă nu pot afecta alți actori decât indirect, prin intermediul mesajelor.

Akka pune la dispoziție un API ușor de folosit pentru crearea de actori și implementarea sistemelor de actori. Tot ce este nevoie pentru a implementa un actor este ca acesta să extindă **trait-ul Actor** și să implementeze metoda abstractă **receive**. Metoda **receive** definește comportamentul actorului în funcție de tipul mesajului primit.

Akka pune de asemenea la dispoziție utilizatorului clasa **Props** ce este folosită pentru a personaliza crearea actorilor și a defini dependințele de care aceștia au nevoie.
Actorii pot fi creați doar pe baza unui **sistem de actori** existent. În cod:

```
implicit val system: ActorSystem = ActorSystem("trafficGenerator", config)
```

Fiecare actor trebuie să aibă un actor supervizor. Actorul supervizor are rolul de a executa regulat "**health-checks**" pentru actorii aflați în supervizarea sa. De altfel când un actor fiu întâlnește o eroare pe care nu o poate trata, aceast va fi transmisă ascendent în ierarhia supervizorilor, până este întâlnit primul subtip de actor capabil de a o trata.

Sistemul de actori va defini un actor supervizor default. Acesta va fi primul nod în ierarhia de supervizori și are definită o metodă abstractă de tratare a erorilor. Utilizatorul are posibilitatea - și este încurajat , de a defini supervizori suplimentari folosind metoda **`watch`** din a instanței inferioare de **`Actor Conext`**, instanțiată implicit la crearea actorilor.

"Deoarece implementarea actorilor este separată de logica internă Akka, eroarea codului tău nu poate afecte sistemul Akka. Când codul tău eșuează, este apelat un set de

[20] Un design pattern reprezintă o soluție generală și reutilizabilă pentru o problemă des întâlnită, ce poate fi aplicată în cadrul dezvoltării unei aplicații software.

pași din Akka ce execută o operație ce cumva tratează excepția. Akka poate rezolva problema codului tău acum în carantină în mai multe moduri, pornind de la mecanismele implicite de tratare a excepțiilor până la implementarea de strategii avansate de error handling definite de către utilizator.” (Wyatt, Kuhn, 6)

Implementarea în cod:

```
protected val osmRouter: ActorRef =
  context.actorOf(FromConfig.props(OSMActor.props(osmRepo, unitGenerator)), name = "osmRouter")
protected val kafkaRouter: ActorRef =
  context.actorOf(FromConfig.props(KafkaActor.props()), name = "kafkaRouter")

implicit val timeout: Timeout = Timeout(10, TimeUnit.SECONDS)

override def preStart(): Unit = {
  context.watch(osmRouter)
  context.watch(kafkaRouter)
}
```

În cadrul Traffic Generator, este creat întâi un sistem de actor numit **trafficGenerator**, iar în cadrul acestuia putem crea actorul supervizor, de tip **`SupervisorActor`**.

SupervisorActor conține logica necesară pentru a superviza funcționalitatea actorilor de tip **`OSMActor`** și **`KafkaActor`**. De asemenea acționează ca un intermediar pentru mesaje, singura logică implementată fiind cea de a trimite mai departe mesajele către subtipul de actori capabili să le proceseze.

```
def receive: Receive = {

  case request: GenerateRequest => sendRequestsToGenerateRoutes(request)

  case ride: Ride => osmRouter forward ride

  case completedStep: CompletedStep => kafkaRouter forward completedStep

  case e: DeadLetter =>
    log.warning(s"The ${e.recipient} is not able to process message: ${e.message
```

Un caz special îl reprezintă cazul **`GenerateRequest`**, pentru care supervizorul generează inițial o listă de rute, compuse dintr-o coordonată de pornire și o coordonată destinație, iar apoi rutele sunt trimise distribuit, în manieră round-robbin către “child actors” de tipul **`OSMActor`** - subtipul OSMActor fiind responsabil cu simularea deplasării în traffic. Se poate observa de altfel că un autovehicul simulat este reprezentat printr-o combinație de listă de puncte reper în care trebuie să ajungă, pas precedent, timpul la care a fost generat pasul precedent, un id generat și un timp rămas . Timpul rămas poate lipsi

deoarece acesta reprezintă faptul că în intervalul de timp de la ultima deplasare autovehiculul ar fi depășit un punct reper și se află într-un punct intermediar, situat pe traseul spre următorul punct reper. Modelul unui autovehicul este reprezentat de *`case class`*-ul, și poate fi interpretat și ca *traseul* pe care acesta îl va parcurge în viitor.

```
case class Ride(
                previousStep: GenerateRouteStep,
                followingSteps: List[GenerateRouteStep],
                previousTime: Double,
                unitId: String,
                remainingTime: Option[Double] = None
              )
```

2.2.5 Kafka Producers

`KafkaActor` este responsabil doar cu publicarea datelor generate de către OSMActori în topicul de Kafka **`completed-steps`**. Acesta extinde **`traitul Actor "mixed-in" cu traitul StepsProducer`**.

Traitul **`StepsProducer`** crează la rândul său un **`KafkaProducer`**. Acesta este implementat sub forma unui **`KafkaProducerActor`**, un actor wrapper peste clasa de **`KafkaProducer`** din driverul classic de Kafka pentru Java. Aceast este importat din proiectul **Scala Kafka Client with Akka**, creat de **cakesolutions**. Este o extensie a driver-ului de Kafka pentru Java. Oferă funcții *helpers* pentru configurarea driverului și oferă de asemenea integrarea cu Akka, tipurile de actori Kafka Producer Actor și Kafka Consumer Actor fiind ideale pentru procesarea performantă, customizabilă și în paralel a stream-urilor de date într-o aplicație bazată pe Akka.

```
lazy val kafkaProducerConf = KafkaProducer.Conf(
  bootstrapServers = appConfig.getString( path = "kafka.bootstrap.servers"),
  keySerializer = new StringSerializer,
  valueSerializer = new JsonSerializer[CompletedStep]
)

lazy val kafkaProducerActor: ActorRef = context.actorOf(KafkaProducerActor.props( kafkaProducerConf))

def submitMsg(topic: String, completedStep: CompletedStep): Unit = {
  kafkaProducerActor ! ProducerRecords(List(KafkaProducerRecord(topic, completedStep.unitId, completedStep)))
}
```

Singurul **`prop`** ce trebuie specificat pentru a crea un **`KafkaProducerActor`** este configurare - de tip **`Conf`**. Pentru use case-ul din Traffic Generator configurarea constă doar în trei parametri. Primul constă în lista bootstrap serverelor, unde un bootstrap server

reprezintă adresa și hostul la care se poate conecta inițial un client Kafka pentru a se conecta la nodurile din cluster, numite și "**brokers**". Ceilalți doi parametri sunt obiecte de tip serializator pentru key recordului de Kafka și pentru valoarea acestuia. Serializatorii convertesc informația stocată în obiecte Scala în byteArray.

```
override def serialize(topic: String, data: A): Array[Byte] =
  stringSerializer.serialize(topic, Json.stringify(Json.toJson(data)))
```

Serializatorul pentru cheie este un simplu **`StringSerializer`** și este oferit de către driverul de bază Kafka. **`JsonSerializer`** este un serializator definit de aplicație, bazat pe **`StringSerializer`** și pe librăriile de formatare JSON din **`Play API`**.

2.2.6 OSM Actor și logica de simulare a traficului

Am specificat anterior că atunci când **`SupervisorActor`** primește un mesaj de tipul generate request - ce semnifică simularea traficului într-o arie definită prin centru și rază pentru un număr dat de unități - **`nbOfUnits`**, acesta generează un traseu aleatoriu generând două puncte la întâmplare în zona dată, pe post de start și destinație, pentru fiecare unit.

Generarea punctelor în interiorul unui cerc de rază **radius** și origine **(latitude, longitude)** se face după formula:

```
val radius = (radiusInKm * 1000) / METERS_TO_DEGREE_RATIO
val r = radius * sqrt(Random.nextDouble())
val angle = Random.nextDouble() * 2 * Pi

val x = origin.latitude + r * cos(angle)
val y = origin.longitude + r * sin(angle)

Coordinate(x, y)
```

Unde **`METERS_TO_DEGREE_RATIO`** reprezintă raportul dintre un grad și un metru, calculată la ecuator.

Algoritmul de deplasare a unei unități este descris în următorii pași:

- Dacă unitate nu mai are pași de parcurs, stop. Altfel treci la pasul următor.

- Se calculează timpul ce a trecut de la ultimul timp înregistrat pentru unitatea în cauză; dacă unitatea are câmpul **`remainingTime`** atunci va fi folosită valoarea acestuia.
- Se generează o viteză aleatoare bazată distanța și timpul estimat furnizate de API-ul OSRM pentru punctul de referință curent.
- Se calculează distanța parcursă de unitate cu formula **d = v * t** și distanța de la punctul curent până la punctul următor folosind **teorema cosinusului pentru triunghiuri sferice**, implementată în librăria **`Geocalc`**.
- Se disting trei cazuri:

    1. Distanța parcursă este aceeași cu distanța până la punctul următor. În acest caz pasul următor devine pasul curent, se apelează funcția de **`updateCompletedSteps`** pe noul pas curent, se updatează ultimul timp înregistrat la timpul curent, se elimină pasul curent din lista pașilor rămași, dacă este cazul se setează **`remainingTime`** la valoarea 0, iar apoi se programează un nou apel al funcției de deplasare după un timp de **`1s + random`**, unde random are valori între 0 și 1.

    2. Distanța parcursă este mai scurtă decât distanța până la următorul punct reper. În acest caz se calculează punctul intermediar în care a ajuns autovehiculul cunoscând distanța parcursă, direcția - trimisă tot de API-ul OSRM și punctul origine, folosind funcția **`pointRadialDistance`** din librăria **`EarthCalc`**, ce calculează punctul aflat la distanța respectivă, în direcția dată, față de punctul de origine. Se marchează punctul curent ca fiind punctul intermediar calculat, punctele următoare nu sunt alterate. Restul pașilor sunt aceeași ca în primul caz.

```
def updateStepsForShorterDistance(traveledDistance: Double, distanceBetweenPoints: Double,
                                  elapsedTime: Double): Ride = {

  val nextStep = ride.followingSteps.head
  val intermediatePoint = CoordinatesGenerator.generateIntermediatePoint(ride.previousStep.location,
    nextStep.location, traveledDistance)
  val updatedDistance = distanceBetweenPoints - traveledDistance
  val updatedPreviousStep = ride.previousStep.copy(location = intermediatePoint, distance = updatedDistance)

  val reachTime = ride.previousTime + elapsedTime
  ride.copy(previousStep = updatedPreviousStep, previousTime = reachTime, remainingTime = None)
}
```

3. Distanța parcursă este mai mare decât distanța până la următorul punct reper. În acest caz se calculează timpul în care autovehiculul ar fi ajuns la punctul reper folosind formula **t = d/v**. Este apelată metoda **`updateCompletedSteps`** pentru punctul reper depășit și timpul setat la ultimul timp înregistrat plus timpul **t** calculat anterior. Se updatează câmpul **`remainingTime`** la valoarea diferenței dintre timpul curent și timpul necesar pentru a ajunge la punctul reper depășit. Se updatează lista pașilor rămași prin eliminarea pasului curent.

- Funcția **`updateCompletedSteps`** este responsabilă pentru publicarea mesajelor cu informații despre pașii completați în topicul **`completed-steps`** din Kafka. Aceasta trimite un mesaj - adică o instanță de case class - către actorul supervizor, acesta urmând să îl redirecționeze către un actor de tip **`KafkaActor`**. Cum am specificat anterior, fiecare conține un **`KafkaProducer`** producer configurat pentru broker-ul de Kafka dorit și topicul **completed-steps**.

Toate aceste funcții sunt definite ca **nested functions** în cadrul funcției **`updateSteps`**.

Integritatea datelor din topicul de completed-steps poate fi verificată folosind comanda predefinită de kafka:

`bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic completed-steps –from-beginning`

```
~/Documents/kafka/kafka_2.12-2.4.0 $ bin/kafka-console-consumer.sh --bootstrap-server localhost:9092 --topic completed-steps --from-beginning
{"unitId":"1222217776467415040","location":{"latitude":8.692137,"longitude":49.414908},"time":1580234400275}
{"unitId":"1222217776467415041","location":{"latitude":8.678011,"longitude":49.427287},"time":1580234400275}
{"unitId":"1222217776572272640","location":{"latitude":8.6666,"longitude":49.409663},"time":1580234400300}
{"unitId":"1222217776391917568","location":{"latitude":8.668558,"longitude":49.405113},"time":1580234400257}
{"unitId":"1222217776735850496","location":{"latitude":8.681092,"longitude":49.417316},"time":1580234400339}
{"unitId":"1222217776748433408","location":{"latitude":8.691794,"longitude":49.426366},"time":1580234400342}
{"unitId":"1222217776475803648","location":{"latitude":8.678011,"longitude":49.427287},"time":1580234400277}
{"unitId":"1222217776790376448","location":{"latitude":8.667844,"longitude":49.420272},"time":1580234400352}
{"unitId":"1222217776404500480","location":{"latitude":8.673588,"longitude":49.421434},"time":1580234400260}
{"unitId":"1222217776391917568","location":{"latitude":8.668558009561016,"longitude":49.40511300017881},"time":1580234400259}
{"unitId":"1222217776794570752","location":{"latitude":8.670869,"longitude":49.417172},"time":1580234400353}
{"unitId":"1222217776224145408","location":{"latitude":8.69406,"longitude":49.408289},"time":1580234400217}
{"unitId":"1222217776337391616","location":{"latitude":8.686061,"longitude":49.401049},"time":1580234400219}
{"unitId":"1222217776735850496","location":{"latitude":8.681092,"longitude":49.417316},"time":1580234400339}
{"unitId":"1222217776748433408","location":{"latitude":8.691794,"longitude":49.42636600000001},"time":1580234400342}
```

Un record din topicul **`completed-steps`** are următoarea structură:

```
{"unitId":"1222269386270838784","location":{"latitude":8.683450847087098,"longitude":49.41735017503189},"time":1580246934084}
```

Aceasta creează un consumer ce afișează recordurile publicate în topicul dat ca parametru, folosind bootstrap brokerul specificat, în cazul de față **`localhost:9092`** în consolă. Astfel putem vizualiza conținutul topicului, ca în exemplul următor:

Este de menționat că numărul instanțelor de actori, atât pentru KafkaActors cât și pentru OSMActors este configurabil prin intermediul fișierului **`application.conf`** . Prin urmare dacă numărul curent de instanțe nu este suficient, utilizatorul poate crește numărul de actori ,și prin urmare viteza de procesare, pentru a se asigura că latența provocată de aplicație nu afectează performanța simulării și intervalul la care pozițiile autovehiculelor sunt actualizate. Este posibilă și implementarea unui număr de actor auto-scalabili, astfel încât să se instanțieze actori suplimentari doar dacă cei deja existenți nu pot face numărului de mesaje primite.

2.3 Traffic Alerts

2.3.1 Prezentare

Modulul **Traffic Alerts** este responsabil cu detectarea zonelor aglomerate în timp real pe baza unui stream cu date de trafic. Acesta consumă date din topicul de kafka **completed-steps** la un interval de timp configurabil - spre exemplu un minut. Bazat pe "pașii" făcuți de vehicule, acesta poate construi zone cu raza egală de asemenea cu o valoare specificată de utilizator și înregistra numărul de autovehicul ce se află în fiecare zonă la momentul respectiv.

2.3.2 Algoritmul de gruparea a autovehiculelor

Traffic Alerts crează aceste zone în manieră **dinamică** folosind următorul procedeu:

- Sunt citiți pașii din topicul de kafka ce au fost publicați în ultimul minut
- Pașii citiți sunt filtrați, astfel că pentru pașii aceluiași autovehicul - pașii ce au același ID, este păstrat doar pasul înregistrat cel mai târziu, întrucât este considerat că acesta este cel mai apropiat de poziția actuală a vehiculului.
- Vehiculele sunt grupate pe baza zonei în care se află, folosind un map. Locația primului vehicul este adăugată în map. Pentru fiecare autovehicul rămas se execută pașii:
- Se iau la rând toate cheile map-ului.

- Dacă locația autovehiculului se află în interiorul ariei circulare ce are ca origine cheia și raza egală cu valoarea configurată de utilizator, atunci aceasta este adăugată la list valorilor ce corespund acestei chei.
- Dacă locația nu se află în raza nici unei chei - zone deja definite, atunci se adaugă o nouă entitate în map cu cheia și valoarea egală cu locația autovehiculului.
- După ce toți pașii citiți au fost procesați se iterează din nou prin toate cheile map-ului și se verifică dacă numărul elementelor din lista de valori a unei chei - adică numărul autoturismelor dintr-o zonă, depășește pragul configurat de către utilizator.
- Pentru zonele în care este depășit acest prag, este publicat un record în topicul **traffic-alerts**, folosind un Kafka producer.

Datele din topicul de Kafka pot fi apoi consumate de către aplicații ce au ca scop oferirea de informații rutiere utilizatorilor, astfel încât aceștia să poată circula cât mai eficient, sau de către servicii ce au ca scop fluidizarea traficului, cum ar fi o aplicație municipală ce poate trimite agenți rutieri care să dirijeze traficul în anumite zone ale orașului.

2.3.3 Serializers și Deserializers

Am văzut în segmentul despre Kafka (*2.2.6 Serializers*) că pentru a putea publica date într-un topic de Kafka este necesar ca acestea să fie serializate într-un șir de bytes. Reversul este adevărat pentru consumul datelor - acestea trebuie traduse dintr-un șir de bytes într-un obiect de tipul dorit, folosind un *Deserializer*.

În cazul Traffic Alerts, tipul de date citit este *CompletedStep*, același tip cu cel al obiectelor publicate de Traffic Generator. Deserializtorul pentru CompletedStep este definit pe baza deserializatorului implicit *StringDeserializer* din driverul de Kafka și *Json format*. *Format[T]* este un tip generic ce creează metode de *serializare* și *deserializare* implicite - vezi Scala implicits în capitolul 2.1.6, pentru tipul *T* folosind conversia la Json.

Implementarea în cod a deserializatorului este următoarea:

```
implicit val coordinateFormat: Format[Coordinate] = Json.format[Coordinate]
implicit val completedStepFormat: Format[CompletedStep] = Json.format[CompletedStep]

class CompletedStepDeserializer extends Deserializer[CompletedStep] {

  override def configure(configs: util.Map[String, _], isKey: Boolean): Unit = Unit

  override def deserialize(topic: String, data: Array[Byte]): CompletedStep = {
    val stringDeserializer = new StringDeserializer
    val result = Json.parse(stringDeserializer.deserialize(topic, data)).as[CompletedStep]
    stringDeserializer.close()
    result
  }
```

Funcția configure este lăsată în mod deliberat goală deoarece aceasta nu este folosită în contextul aplicației, însă nu poate fi omisă deoarece este definită în traitul *Deserializer[T]*.

Este de observat că deoarece tipul *CompletedStep* conține un câmp cu tipul complex *Coordinate*, este necesar să definim un *Json format* și pentru acest tip.

Serializatorul - folosit pentru publicarea recordurilor de tip *HeavyTrafficArea* este un tip implicit deoarece este comun ambelor module: Traffic Alerts și Traffic Density și este folosit pentru serializarea unui număr mare de tipuri de recorduri. Prin contrast, citirea din Kafka se face pentru un singur record și prin urmare nu este necesară definirea unui deserializator generic. Implementarea serializatorului este următoarea:

```
implicit val coordinateFormat: Format[Coordinate] = Json.format[Coordinate]
implicit val heavyTrafficAreaFormat: Format[HeavyTrafficArea] = Json.format[HeavyTrafficArea]
implicit val coordinateDensityPairFormat: Format[(Coordinate, Long)] = Json.format[(Coordinate, Long)]
implicit val areaDensityFormat: Format[AreaDensity] = Json.format[AreaDensity]

class JsonSerializer[A: Writes] extends Serializer[A] {

  private val stringSerializer = new StringSerializer

  override def configure(configs: util.Map[String, _], isKey: Boolean): Unit =
    stringSerializer.configure(configs, isKey)

  override def serialize(topic: String, data: A): Array[Byte] =
    stringSerializer.serialize(topic, Json.stringify(Json.toJson(data)))

  override def close(): Unit =
    stringSerializer.close()
}
```

Implementarea este totuși asemănătoare cu cea a *Deserializerului* datorită faptului că traitul *Serializer[T]* este de asemenea generic.

Case classul folosit pentru a reprezenta recordurile ce vor fi publicate în Kafka este:

```
case class HeavyTrafficArea(
                             center: Coordinate,
                             radius: Double,
                             nbOfUnits: Long,
                             time: Long
                           )
```

2.3.4 Procesarea cu Spark Streaming

Datele sunt consumate din topicul de kafka în manieră streaming, folosind motorul **Spark**. Spark-ul permite citirea datelor în manieră streaming din surse multiple de date, una dintre aceasta fiind Kafka.

```
val sparkConfig = new SparkConf().setMaster("local[*]").setAppName("Traffic Alerts")
val sparkStreamingContext = new StreamingContext(sparkConfig, Seconds(updateTime))
```

Primul pas în flow-ul aplicație constă în definirea și inițializarea unui **Spark Streaming Context**. Streaming Contextul reprezintă punctul de acces la funcționalitățile Spark. Acesta pune la dispoziție metode ce permit crearea de obiecte de tipul **`DStream`** ce cu Kafka setat ca sursă de input. Spark contextul este inițializat pe baza unei configurații de tipul **`SparkConf`** ce conține **Spark master URL** - un host și port unde poate fi creată interfața Spark de supraveghere a stream-urilor, și numele aplicației.

Pe lângă configurație streaming contextul primește ca parametru intervalul de timp la care va citi date. Traffic Alerts permite utilizatorului să configureze numărul de secunde pentru acest interval.

"Conexiunea cu Kafka se face folosind librăria specializată pentru kafka din toolset-ul Spark Streaming: **`org.apache.spark.streaming.kafka010.`**

Pentru citire este necesar crearea unei configurații pentru Kafka ce cuprinde: modul de conectare la clusterul de Kafka - în cazul de față fiind folosită metoda de conectare prin **bootstrap servers**; clase deserializator[21] pentru cheia și valoarea recordurilor Kafka, identificatorul grupului de consumeri și lista topic-urilor din care se va citi." (Documenație oficială, 9)

[21] Deserializarea reprezintă procedeul de translatarea a unui `byteArray` în obiecte de tipul clasei dorite din program.

Stream-ul de citire este de tip **`DStream`** și este creat prin metoda utilitară **`createDirectStream`**. DStream-ul este apoi transformat dintr-un DStream de record-uri Kafka într-un DStream de pași.

Fiecare set de date citit într-un interval de timp este convertit într-un **RDD**. Datele RDD-ului sunt filtrate astfel încât pentru fiecare autovehicul se păstrează doar cea mai recentă poziție. Acestea sunt grupate pe zone prin următorul set de pași: datele sunt grupate după id-ul autovehiculului folosind **`groupBy(_.id)`**, apoi sunt grupate după zona din care fac parte folosind funcția **`aggregate`**. Aceasta primește ca parametri elementul neutru - elementul de la baza operației de reduce, în cazul de față un map gol; funcția **`seqOp`** - ce definește operația ce se va aplica pe rezultatul agregărilor precedente și elementul curent; și funcția **`combOp`** ce definește operația de îmbinate a rezultatelor obținute din agregarea elementelor partițiilor. Funcția **`combOp`** este necesară deoarece metoda aggregate nu este aplicată secvențial pe întreg RDD-ul. RDD-ul este inițial partiționat iar apoi fiecare partiție este procesată în paralel. La ultimul pas rezultatele parțiale sunt îmbinate folosind funcția **`combOp`**. Funcția constă în unirea cheilor map-urilor și concatenarea listelor de valori pentru cheile ce se găsesc în ambele.

```
stepsStream.foreachRDD(stepsRDD => {

  //for every unit, keep only the latest reached step
  val latestSteps = stepsRDD.groupBy(_.unitId).mapValues(_.maxBy(_.time)).values

  //aggregate steps by area
  val aggregatedSteps =
    latestSteps.aggregate(Map[CompletedStep, Long]())((map, step) => groupPointsByRadius(radius)(map, step),
      (map1, map2) => combineMapsByRadius(radius)(map1, map2))

  //transform completed steps to locations
  val areaTraffic = aggregatedSteps.map { case (key: CompletedStep, value: Long) => (key.location, value) }

  recordHeavyTraffic(areaTraffic)
```

Funcția **`recordHeavyTraffic`** este responsabilă pentru crearea de Kafka records din entry-urile map-ului de unități per arie în recorduri de Kafka și publicarea lor în topicul **`traffic-alerts`**. Spre deosebire de Traffic Generator, publicarea se face folosind un **KafkaProducer** clasic, nu un KafkaProducerActor.

Datele din topicul **`traffic-alerts`** au următoarea structură:

`{"center":{"latitude":8.680577718356037,"longitude":49.412059310887926},"radius":100,"nbOfUnits":5,"time":1580246462876}`

Recordurile **`traffic-alerts`** pot fi vizualizate în consolă folosind console consumerul:

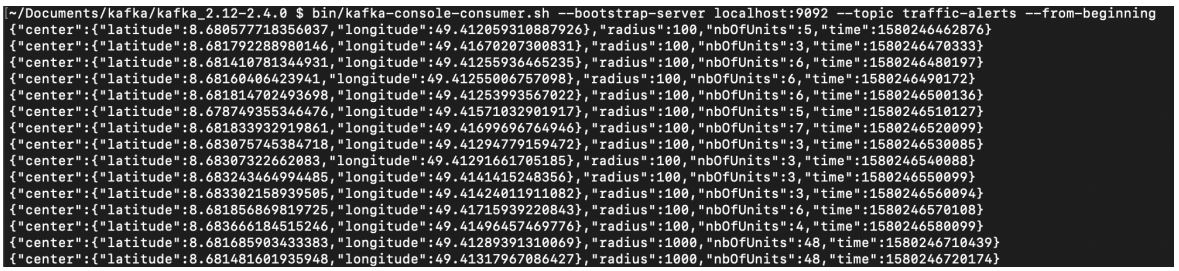

fig 2.2 Recordurile din topicul Kafka "traffic-alerts"

Datele vor persista în Kafka pentru perioada de timp configurată de către dezvoltatorul aplicației. Perioada default este de 7 zile, însă ținând cont că alertele de trafic sunt trimise la un interval de timp configurabil, putem modela perioada de persistență a datelor în funcție de acest interval pentru a evita stocarea datelor redundante.

Procesarea este începută folosind funcția `sparkStreamingContext.start`. Aceasta semnalizează instanței Spark să înceapă citirea datelor din Stream și procesarea RDD-urilor citite la fiecare interval de timp prin operațiile de transformare și acțiunile definite în funcția dată ca parametru pentru `foreachRDD`.

2.4 Traffic Density

Modulul **Traffic Density** este responsabil cu detectarea celor mai aglomerate subarii din interiorul unei arii definite de utilizator. Aria este definită ca un set de 4 valori reprezentând latitudinea minimă, latitudinea maximă, longitudinea minimă și longitudinea maximă.

2.4.1 Configurare

Utilizatorul trebuie să definească latura subariei, subariile având formă pătrată.

De altfel, utilizatorul trebuie să configureze intervalul de timp pentru care se dorește calcularea subariilor. Limitele intervalului sunt reprezentate de ziua calendaristică și timpul cu precizie până la secundă, formatul acesteia fiind **`yyyy-MM-dd HH:mm:ss`**.

Pentru conectarea la Kafka este necesară specificarea partiției și intervalul offset-urilor de la care se dorește citirea. Spre exemplu pentru un offset de început de 1000 și un offset de final de 2000, aplicația va consuma recorduri de la al 1000-lea record

publicat în partiția dată până la recordul al 2000-lea. Două comenzi utile pentru determinarea offset-urilor sunt:

- comanda pentru determinarea celui mai vechi offset prezent în topic:

```
bin/kafka-run-class.sh kafka.tools.GetOffsetShell --broker-list localhost:9092 --topic mytopic --time -2
```

- comanda pentru determinarea celui mai nou offset prezent în topic:

```
bin/kafka-run-class.sh kafka.tools.GetOffsetShell --broker-list localhost:9092 --topic mytopic --time -1
```

Traffic Density se conectează la clusterul de Kafka similar cu modulul **Traffic Alerts**. Aceste se folosește de **Spark Streaming** și librăria dedicată a acestuia pentru conectarea la Kafka și citirea Kafka record-urilor: `org.apache.spark.streaming.kafka010`.

Configurarea Kafka pe baza căreia se citesc datele este similară cu cea din Traffic Alerts, precizând adresa broker-ilor, clasele deserializator și id-ul grupului de consumers. Spre deosebire de Traffic Alerts, citirea se face în manieră **batch**, nu în manieră streaming. Asta înseamnă că toate recordurile din intervalul de offset-uri dat sunt consumate simultan, operația de citire executându-se doar o singură dată.

Pe când în cazul citirii pe bază de streaming recordurile erau citite într-un **DStream**, pe baza căruia era creat un **RDD** separat pentru datele consumate la fiecare interval de timp, în cazul citirii batch datele sunt stocate direct într-un **RDD**. După citirea lor datele sunt transformate din recorduri în instanțe de clase Scala apelând metoda `map` pe RDD.

2.4.2 Crearea map-ului de subarii

Pe lângă crearea RDD-ului, la pornirea aplicației sunt create și subariile zonei definită în configurare. Logica de creare a acestora este următoarea:

- Se generează colțurile ariei, reprezentate prin coordonate geografice de forma (latitudine, longitudine);

```
val stepsRDD = KafkaUtils.createRDD[String, CompletedStep](
  sparkContext,
  kafkaConfig.asJava,
  offsetRanges,
  LocationStrategies.PreferConsistent)
  .map { stepRecord => stepRecord.value }
```

- Pornind de colțul din stânga-sus se generează subarii pătrate, cu latura de dimensiunea configurată, pe direcție verticală, descendent, până se ajunge la colțul din jos-stânga;
- Se generează pătrate în manieră similară pentru partea dreaptă;
- Pentru fiecare pătrat din coloana stângă, se generează pătrate până se ajunge la pătratul de pe aceeași linie din coloana dreaptă - adică se deplasează păstrând latitudinea constantă.
- Se întoarce lista subariilor împreună cu centrul subariei nou-formate. De precizat este că subariile sunt memorate doar sub forma centrului acestora, întrucât latura este o constantă cunoscută.

Va fi creat ulterior un map cu cheile setate la centrele subariilor create. Astfel, pentru fiecare subarie vom putea înregistra mișcările - sau "pașii", autovehiculelor ce au circulat în interiorul acestora în intervalul de timp dat. Am ales folosirea numărului de mișcări deoarece am considerat că o mașină afectează traficul cu o pondere mai mare cu cât circulă mai îndelungat. Spre exemplu un drum de 500 m va afecta traficul mai puțin decât un drum de 2 km.

Pentru a eficientiza maparea fiecărui pas la subaria în care a avut loc, am definit un **hash map** ce are ca valori centrele subariilor, iar ca **funcție de hashing** am folosit distanța față de centrul ariei date exprimată în laturi de lungimea configurată. Cu alte cuvinte pentru o latură de pătrat configurată cu lungimea **l**, funcția de hashing pentru un punct **p** și centrul **o** va fi parte **[distanța(p, o) / l]**, unde **[ ]** este partea întreagă.

Beneficiul hash map este că oferă căutarea rapidă a subariei de care aparține un pas fără a trebui updatat. O dată calculat acesta este constant. Prin urmare poate fi folosit pentru maparea **în paralel** a pașilor din **RDD** la subariile create.

Subariile sunt generate folosind funcția:

```
getSubAreasCenter(start: Point, end: Point, step: Double,
        stopCondition: (Point, Point) => Boolean): List[Point]
```

Aceasta este o funcție recursivă și de asemenea este de tipul high-order-function. Criteriul de oprire al operației recursive este dat ca o funcție ce primește două puncte și întoarce un boolean. Astfel putem refolosi funcția `getSubAreasCenter` pentru a genera pătrate pe direcție verticală, condiția de oprire fiind dacă latitudinea centrului ariei generate este mai mare decât cea a punctului de oprire, iar pentru generarea pe direcție orizontală verificăm dacă latitudinea punctului generat o depășește pe cea a punctului de sfârșit.

```
def verticalStopCondition(lower: Point, upper: Point): Boolean = lower.getLatitude >= upper.getLatitude

def horizontalStopCondition(left: Point, right: Point): Boolean = left.getLongitude >= right.getLongitude

val leftColumn = getSubAreasCenters(upperLeftCorner, lowerLeftCorner, step, verticalStopCondition)
val rightColumn = leftColumn.map(point => Point.build(point.getLatitude, eastExtreme))

val subAreasCenters = (leftColumn, rightColumn).zipped.toList.flatMap { ends =>
  getSubAreasCenters(ends._1, ends._2, step, horizontalStopCondition)
}
```

2.4.3 Procesarea Recordurilor

Procesarea recordurilor se face pe baza funcției **aggregate**, asemeni procesării din Traffic Alerts. Elementul zero va fi un map gol, al căror chei vor fi centrele subariilor. Funcția de agregare a elementelor (**`seqOp`**) va consta în doi pași. Primul este calcularea funcției hash pentru locația la care a avut loc pasul și apoi obținerea subariilor din hash map-ul de distanțe ce corespund rezultatului. Al doilea pas constă în iterarea listei de subarii obținută la pasul anterior și determinarea centrului de subarie aflat la distanță minimă față de locația pasului. După ce a fost determinată subaria de care aparține pasul, map-ul de agregare va fi updatat cu un entry ce are ca cheie centrul subariei determinate și ca valoare numărul precedent de pași înregistrat pentru aceasta plus un pas.

Funcția de combinare a map-urilor de agregare (**`combOp`**) este aceeași ca în cazul Traffic Alerts, anume reuniunea cheilor din fiecare rezultat parțial, iar pentru cheile comune ambelor map-uri se crează un entry cu cheia respectivă și suma valorilor din ambele map-uri.

Implementarea în cod a funcției aggregate este următoarea:

```
val subAreaDensities = stepsRDD
  .filter(step => inTimeWindow(startTime, endTime, step.time))
  .aggregate(toEmptyMap(subAreas))((map, step) =>
    addEntryToSubareasMap(toPoint(step), map, subAreasHashMap, areaCenter, side),
    (map1, map2) => combineSubAreasMaps(map1, map2))
```

Elementul zero este un map gol creat de funcția **`toEmptyMap`**, funcția **`addEntryToSubAreaMap`** este funcția **`seqOp`**, iar funcția **`combOp`** este **`combineSubAreasMap`**.

Implementarea acestor două funcții este următoarea:

```
def addEntryToSubareasMap(point: Point, subareasMap: Map[Point, Long], subAreasHashMap: Map[Int, List[Point]],
                          center: Point, step: Double): Map[Point, Long] = {

  val distanceFromCenter = nbOfStepsFromCenter(point, center, step)
  if (subAreasHashMap.contains(distanceFromCenter)) {
    val possibleSubAreas = subAreasHashMap(nbOfStepsFromCenter(point, center, step))
    val subArea = possibleSubAreas.minBy(EarthCalc.getDistance(_, point))
    subareasMap + (subArea -> (subareasMap(subArea) + 1))
  }
  else subareasMap
}

def combineSubAreasMaps(subAreaMap1: Map[Point, Long], subAreaMap2: Map[Point, Long]): Map[Point, Long] = {

  val merged: Seq[(Point, Long)] = subAreaMap1.toSeq ++ subAreaMap2.toSeq
  val grouped: Map[Point, Seq[(Point, Long)]] = merged.groupBy(_._1)
  val cleaned: Map[Point, List[Long]] = grouped.mapValues(_.map(_._2).toList)
  cleaned.mapValues(_.sum)
}
```

După ce s-a creat map-ul densității traficului per subarie acesta este trimis către o instanță KafkaProducer și este public în topicul **traffic-density.** După această operație programul își încheie execuția.

Recordurile publicate în Kafka pot fi vizualizate folosind console consumerul. Structura recordului este următoarea: extremă nordică, extremă sudică, extremă estică, extremă vestică - aceste câmpuri definesc aria pentru care s-a calculat densitatea; începutul și sfârșitul intervalului de timp pentru care a fost calculată densitatea, în format **`yyyy-MM-dd hh:mm:ss`**; lungimea laturii pătratului și o listă de perechi, unde primul element al perechii este centrul subariei și al doilea este numărul de acțiuni întreprinse în cadrul acestuia. Un exemple de astfel de record este:


```
{“north”:8.6815,"south":8.681,"east":49.4215,"west":49.41,"side
length: 100m”, ”startTime”:"2020-01-21
22:32:00","endTime":"2020-01-21
22:33:00”,”subAreaDensities”:[{“_1”:{“latitude”:8.681500054028577,
”longitude":49.414093845390475},"_2":76},{"_1":{"latitude":8.68150
0162085731,"longitude":49.42228153617321},"_2":192},{"_1":{"latitu
de":8.681500108057154,"longitude":49.41818769078156},"_2":15},{"_1
":{"latitude":8.681500072038103,"longitude":49.415458460520775},"_
2":84},{"_1":{"latitude":8.681500036019052,"longitude":49.41272923
026024},"_2":131},{"_1":{"latitude":8.6815,"longitude":49.41},"_2"
:55},{"_1":{"latitude":8.681500126066679,"longitude":49.4195523059
1204},"_2":59},{"_1":{"latitude":8.681500090047628,"longitude":49.
41682307565112},"_2":87},{"_1":{"latitude":8.681500144076205,"long
itude":49.42091692104259},"_2":5},{"_1":{"latitude":8.681500018009
526,"longitude":49.41136461513008},"_2":18}]}
```

2.4.4 Modele folosite și fișierul de configurare

Acesta este modelat în aplicația Traffic Density cu case class-ul **`AreaDensity`**.

```
case class AreaDensity(
                        north: Double,
                        south: Double,
                        east: Double,
                        west: Double,
                        startTime: String,
                        endTime: String,
                        subAreaDensities: List[(Coordinate, Long)]
                      )
```

Configurarea modului Traffic Density se află în fișierul *application.conf* și are structura următoare:

```
trafficDensity {

    side = 150 //in meters

    startTime = "2020-01-21 22:32:00" //must be in format yyyy-MM-dd HH:mm:ss
    endTime = "2020-01-21 22:33:00"

    partitions = [0]
    fromOffsets = [1200]
    toOffsets = [1500]

    northExtreme = 8.6815
    southExtreme = 8.681
    westExtreme = 49.41
    eastExtreme = 49.4215

    kafka {
        bootstrap.server = "localhost:9092"

        input {
          topic = "completed-steps"
        }
        output {
          topic = "area-density"
        }
    }
}
```

Concluzii

Fluidizarea traficului reprezintă o problemă cu care se confruntă majoritatea marilor orașe. Un procent semnificativ din populația acestora conduc mașina personală aproape zilnic, iar infrastructura rutieră nu poate face față numărului de autovehicule în stadiul curent. Aceasta trebuie constant modernizată și actualizată în funcție de nevoile cetățenilor.

Un număr important din persoanele ce utilizează des autoturismul personal folosesc de asemenea o aplicație mobilă pentru a naviga prin trafic, cum ar fi Google Maps sau Waze. Acestea la rândul lor colectează date anonime despre vehicule, cum ar fi poziția și viteza acesta.

Traffic Analyzer poate ajuta la fluidizarea traficului folosind procedee de analiză big data pentru a indica zonele supraaglomerate și transmiterea acestei informații părților interesate.

Serviciul Traffic Alerts este un serviciu de notificări în timp real ce informează utilizatorul în ce zone din oraș se circulă îngreunat, oferindu-i acestuia posibilitatea de a evita aceste zone și de a-și configura traseul în timp real.

Serviciul Traffic Density oferă informații privind nivelul de aglomerare al sectoarelor de oraș pe baza datelor istorice. Putând configura atât aria analizată cât și perioada de timp studiată, serviciul oferă utilizatorilor informații utile pentru plănuirea călătoriilor de lungă durată. Spre exemplu, dacă utilizatorul începe călătoria la ora 09:00, iar Traffic Density îl informează că zona Aurel Vlaicu este supraaglomerată în intervalul de timp 10 - 12, acesta își poate configura traseul în prealabil astfel încât să ocolească această zonă. Această decizie nu ar fi putut fi luată pe baza Traffic Alerts deoarece când serviciul Traffic Alerts ar fi trimis notificarea traficul din zona Aurel Vlaicu ar fi fost deja afectat, iar utilizatorul posibil fie să fi fost deja participant la acesta, fie să nu mai considere fiabilă opțiunea reconfigurării traseului.

De aceste servicii pot beneficia de asemenea și companiile de asigurări sau cele de închirieri mașini. Cunoscând tiparele traficului pentru zone configurabile, acestea pot modifica prețurile în funcție de factorii de risc. Spre exemplu, dacă folosind Traffic Density în perioada sărbătorilor de iarnă, pentru un oraș aleatoriu, se ajunge la concluzia că traficul în acel oraș traficul este mai aglomerat și prin urmare mașina va prezinta un risc sporit de a fi avariată dacă va fi condusă în zona respectivă, o companie de închirieri de

autovehicule poate crește prețurile pentru orașul în cauză pentru a-și acoperi potențialele daune.

Atât procesare în timp real a streamurilor de date folosită în Traffic Alerts, cât și procesarea cantităților foarte mari de date istorice simultan, în paralel necesară pentru funcționarea Traffic Density este posibilă datorită motorului de procesare Spark, iar stocarea și citirea datelor în manieră distribuită este posibilă datorită platformei Kafka.

Scala este limbajul ideal pentru dezvoltarea acestor servicii deoarece este un limbaj funcțional și modern, ce permite dezvoltarea de aplicații în Spark cât și conectarea la Kafka prin intermediul driverelor de Java la care are acces.

Traffic Analyzer este o soluție modernă la o problemă veche, ce există de zeci de ani, cea a blocajelor de trafic. Aceasta se folosește de datele furnizate chiar de participanții la trafic pentru a le permite să circule eficient și să economisească timp.

Bibliografie

[1] Alvin Alexander. Functional Programming Simplified, 2017

[2] Paul Chiusano, Rúnar Bjarnason. Functional Programming in Scala, 2014

[3] Bill Venners, Martin Odersky. Programming in Scala, 2008

[4] Holden Karau, Matei Zaharia, Andy Konwinski, Patrick Wendell. Learning Spark: Lightning-Fast Big Data Analysis, 2015

[5] Gwen Shapira, Neha Narkhede, and Todd Palino. Kafka: The Definitive Guide: Real-Time Data and Stream Processing at Scale, 2017

[6] Derek Wyatt. Akka Concurrency: Building Reliable Software in a Multi-core World, 2013

[7] https://kafka.apache.org/quickstart

[8] https://www.coursera.org/learn/scala-spark-big-data?

[9] https://spark.apache.org/docs/2.2.0/streaming-kafka-0-10-integration.html