

# Sisteme Distribuite - Laborator 13

## Apache Kafka și Apache Spark

### Introducere

#### Apache Kafka

Apache Kafka este o platformă de fluxuri de evenimente distribuite, capabilă să gestioneze miliarde de evenimente pe zi. Această platformă poate:

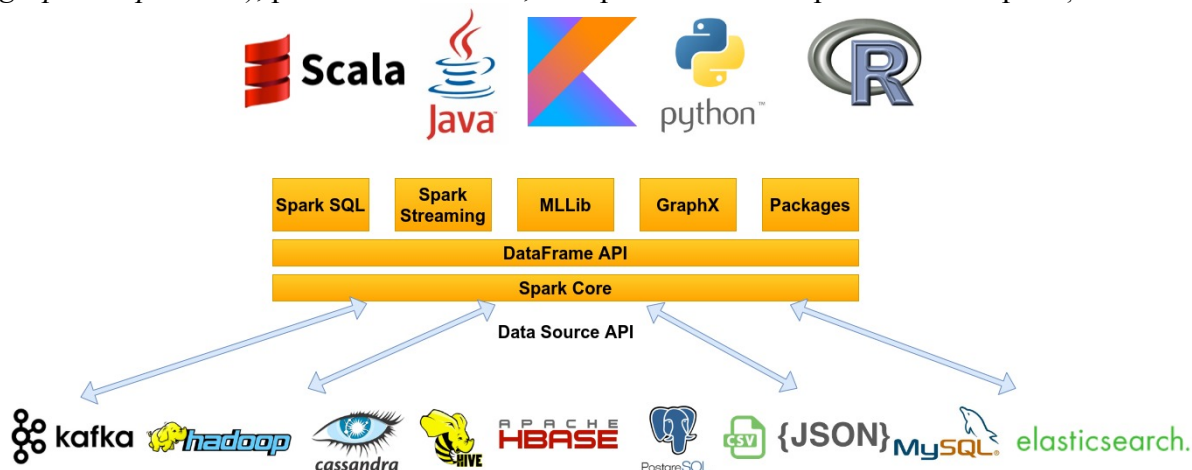
- să publice/să se aboneze la fluxuri de înregistrări (eng. *records*), în mod similar cu o coadă de mesaje;
- să stocheze fluxuri de înregistrări într-un mod durabil și tolerant la erori (*fault-tolerant*)
- să proceseze fluxuri de înregistrări pe măsură ce apar

Pentru mai multe detalii, vezi:

- [Cursurile de la disciplina Sisteme distribuite](#)
- [Laboratorul 10 de la disciplina Sisteme distribuite](#)
- <https://www.confluent.io/what-is-apache-kafka/>
- <https://kafka.apache.org/intro>
- <https://kafka.apache.org/documentation/>

#### Apache Spark

Spark este un motor de uz general de procesare distribuită a datelor. Peste *framework*-ul Spark, există biblioteci pentru SQL, învățare automată (*machine learning*), calcule pe grafuri (*graph computation*), procesări de fluxuri, care pot fi folosite împreună într-o aplicație.



#### Cazuri tipice de utilizare:

- **Procesarea fluxurilor (*stream processing*):** deși este fezabilă stocarea datelor pe disk și procesarea lor ulterioară, uneori este necesar să se acționeze asupra datelor pe măsură ce acestea sosesc (spre exemplu tranzacțiile financiare trebuie procesate în timp real pentru a identifica și refuza tranzacții potențial frauduloase)
- **Învățare automată (*machine learning*):** Soluțiile software pot fi antrenate să identifice și să acționeze în funcție de trigger-ele din seturile de date bine înțelese, înainte de a aplica aceleași soluții la date noi și la date necunoscute. Abilitatea *framework*-ului Spark de a stoca date în memorie și de a executa rapid interogări

repetate îl face o alegere potrivită pentru antrenarea algoritmilor de învățare automată.

- **Analiză interactivă (*interactive analytics*):** procesul de interogare interactivă necesită un sistem precum Spark care să fie capabil să răspundă și să se adapteze rapid.
- **Integrarea datelor (*data integration*):** *Spark* și *Hadoop* sunt folosite din ce în ce mai mult pentru a reduce costul și timpul necesar pentru procesul **ETL** (*Extract, Transform, Load*)

#### *Avantaje:*

- Simplitate
- Viteză
- Suport
- *Pipeline*-uri de date

Pentru mai multe detalii despre framework-ul Spark, precum și istoria acestuia (de ce s-a trecut de la Hadoop la Spark), vezi:

- Cursurile de la disciplina *Sisteme distribuite*
- <https://mapr.com/blog/spark-101-what-it-what-it-does-and-why-it-matters/>
- <https://spark.apache.org/docs/2.4.5/>

## Spark RDD (Resilient Distributed Dataset)

Abstractizarea principală oferită de Spark este un **RDD**, mai precis, o colecție de elemente partiționate pe nodurile dintr-un cluster pe care se poate acționa în paralel.

O altă abstractizare în Spark reprezintă **variabilele partajate** care pot fi utilizate în operații paralele. În mod implicit, când Spark execută o funcție în paralel ca un set de task-uri pe diferite noduri, furnizează o copie a fiecărei variabile utilizate în funcție către fiecare task. Uneori, este nevoie totuși de partajarea unei variabile între task-uri. Framework-ul Spark suportă două tipuri de variabile partajate: *variabile de broadcast* care pot fi utilizate pentru păstrarea în cache a unei variabile în memorie pe toate nodurile, și *acumulatori* care sunt variabile în care doar se adaugă (*counter*, *sume*).

**Pentru mai multe detalii, vezi:**

- <https://spark.apache.org/docs/2.4.5/rdd-programming-guide.html>

### *Crearea unui proiect Spark (RDD)*

Se creează un nou **proiect Maven**, se bifează opțiunea „**Create from archetype**” și se selectează: **org.jetbrains.kotlin:kotlin-archetype-jvm** → *kotlin-archetype-jvm:1.3.72*. Se adaugă apoi următoarea dependență în *pom.xml*:

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-core_2.12</artifactId>
  <version>2.4.5</version>
</dependency>
```

Este nevoie de crearea unui obiect **JavaSparkContext** care îi spune *framework*-ului Spark cum să acceseze un cluster.

**Observație:** Pentru simplitate, se va configura Spark să fie executat local. Pentru o imagine de ansamblu asupra configurării Spark, vezi:

- <https://spark.apache.org/docs/2.4.5/submitting-applications.html>

## Spark SQL

Spark SQL este un modul al *framework*-ului Spark pentru procesarea datelor structurate. O utilizare a Spark SQL este pentru a executa interogări SQL. Rezultatele returnate în urma unei interogări vor fi *Dataset-uri* sau *DataFrame-uri*.

*Dataset*-ul este o colecție distribuită de date, ce are atât avantajele unui RDD (funcții lambda) cât și pe cele ale motorului de execuție optimizat al Spark SQL.

Un *DataFrame* este un *Dataset* organizat în coloane cu denumiri. Conceptual, este echivalent cu o tabelă dintr-o bază de date relațională, dar cu mai multe optimizări.

**Pentru mai multe detalii, vezi:**

- <https://spark.apache.org/docs/2.4.5/sql-getting-started.html>

**De asemenea, vezi funcțiile de agregare pentru DataFrame-uri:**

- [https://spark.apache.org/docs/2.4.5/api/scala/index.html#org.apache.spark.sql.functions\\$](https://spark.apache.org/docs/2.4.5/api/scala/index.html#org.apache.spark.sql.functions$)

### Adăugarea componentei Spark SQL

Pentru a include componenta Spark SQL în proiect, este necesară adăugarea următoarei dependențe în fișierul *pom.xml*:

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-sql_2.12</artifactId>
  <version>2.4.5</version>
</dependency>
```

## Securitate

Securitatea este dezactivată în mod implicit în Spark. Acest lucru înseamnă că, în mod implicit, aplicația este vulnerabilă la atacuri.

*Framework*-ul Spark poate fi securizat prin următoarele aspecte:

- Spark RPC (protocolul de comunicare între procese Spark)
- Criptarea spațiului de stocare local (*local storage encryption*)
- Web UI
- Configurarea porturilor pentru securitatea rețelei
- Kerberos
- Jurnal de evenimente (*event logging*)

```
val sparkConf = SparkConf().setMaster("local[4]")
                              .setAppName("Spark Example")
                              .set("spark.io.encryption.enabled", "true")
                              .set("spark.io.encryption.keySizeBits", "256")
val sparkContext = JavaSparkContext(sparkConf)
```

**Pentru mai multe detalii privind opțiunile de securizare ale unei aplicații Spark, precum și modul în care pot fi adăugate aceste configurări, vezi:**

- <https://spark.apache.org/docs/latest/configuration.html>
- <https://spark.apache.org/docs/latest/security.html>

**Pentru securizarea Apache Kafka, vezi documentația oficială, capitolul 7:**

- <https://kafka.apache.org/documentation/#security>

## Spark Streaming

*Spark Streaming* primește fluxuri de date în timp real, le împarte în loturi (eng. *batches*) care sunt procesate apoi de motorul Spark (*Spark Engine*) pentru a genera fluxul final de rezultate, tot sub formă de loturi (*batches*).



Modul de funcționare al Spark Streaming

Spark Streaming oferă o abstractizare de nivel înalt numită *flux discret* (*discretized stream*) sau *DStream*, ce reprezintă un flux continuu de date. *DStream*-urile pot fi create fie din fluxuri de date de intrare din surse precum *Kafka*, *Flume* și *Kinesis*, fie aplicând operațiuni de nivel înalt pe alte *DStream*-uri. Un *DStream* este reprezentat intern ca o secvență de RDD (abstractizarea Spark-ului pentru un set de date imutabil distribuit).

**ATENȚIE:** Un program *Spark Streaming* executat local trebuie să utilizeze în URL-ul master "**local[n]**", unde  $n \geq 2$ , deoarece un *thread* va executa *receiver*-ul, iar restul vor procesa datele primite.

Există mai multe surse de date pentru *Spark Streaming*:

- TCP:

```
val lines: JavaReceiverInputDStream<String> =
  streamingContext.socketTextStream("localhost", 9999)
```

- fișiere text:

```
streamingContext.textFileStream(dataDirectory)
```

- flux de fișiere:

```
streamingContext.fileStream<KeyClass, ValueClass,
  InputFormatClass>(dataDirectory)
```

- surse avansate: *Kafka*, *Flume*, *Kinesis*

**Pentru mai multe detalii, vezi:**

- <https://spark.apache.org/docs/2.4.5/streaming-programming-guide.html> (în special transformările pe *DStream*-uri și operațiile de output pe *DStream*-uri)
- <https://spark.apache.org/docs/2.4.5/streaming-custom-receivers.html>

### Adăugarea componentei Spark Streaming

Pentru a include componenta *Spark Streaming* în proiect, este necesară adăugarea următoarei dependențe în fișierul *pom.xml*:

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-streaming_2.12</artifactId>
  <version>2.4.5</version>
</dependency>
```

## Integrarea Spark Streaming cu Kafka

Se va adăuga următoarea dependență în fișierul *pom.xml*:

```
<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-streaming-kafka-0-10_2.12</artifactId>
  <version>2.4.5</version>
</dependency>
```

Pentru mai multe detalii, vezi:

- <https://spark.apache.org/docs/2.4.5/streaming-kafka-0-10-integration.html>

## Exemple

### Exemplul 1: Spark RDD

Vezi comentariile cu explicații din cod.

```
package com.sd.laborator

import org.apache.spark.SparkConf
import org.apache.spark.api.java.JavaSparkContext
import org.apache.spark.api.java.function.Function
import org.apache.spark.api.java.function.Function2
import org.apache.spark.broadcast.Broadcast
import org.apache.spark.storage.StorageLevel

internal class GetLength : Function<String?, Int?> {
  override fun call(p0: String?): Int? {
    return p0?.length ?: 0
  }
}

internal class Sum : Function2<Int?, Int?, Int?> {
  override fun call(p0: Int?, p1: Int?): Int? {
    return (p0?.toInt() ?: 0) + (p1?.toInt() ?: 0)
  }
}

fun main(args: Array<String>) {
  // configurarea Spark
  val sparkConf = SparkConf().setMaster("local").setAppName("Spark
Example")
  // initializarea contextului Spark
  val sparkContext = JavaSparkContext(sparkConf)

  val items = listOf("123/643/7563/2134/ALPHA",
"2343/6356/BETA/2342/12", "23423/656/343")
  // paralelizarea colectiilor
  val distributedDataset = sparkContext.parallelize(items)

  // 1) spargerea fiecarui string din lista intr-o lista de
substring-uri si reunirea intr-o singura lista
  // 2) filtrarea cu regex pentru a pastra doar numerele
```

```

    // 3) conversia string-urilor filtrate la int prin functia de
mapare
    // 4) sumarea tuturor numerelor prin functia de reducere
    val sumOfNumbers = distributedDataset.flatMap {
it.split("/").iterator() }
        .filter { it.matches(Regex("[0-9]+")) }
        .map { it.toInt() }
        .reduce {total, next -> total + next }
    println(sumOfNumbers)

    // seturi de date externe
    // setul de date nu este inca incarcat in memorie (si nu se
actioneaza inca asupra lui)
    val lines = sparkContext.textFile("src/main/resources/data.txt")

    // pentru utilizarea unui RDD de mai multe ori, trebuie apelata
metoda persist:
    lines.persist(StorageLevel.MEMORY_ONLY())

    // functia de mapare reprezinta o transformare a setului de date
initial (nu este calculat imediat)
    // abia cand se ajunge la functia de reducere (care este o
actiune) Spark imparte operatiile in task-uri
    // pentru a fi rulate pe masini separate (fiecare masina executand
o parte din map si reduce)
    // exemplu cu functii lambda:
    val totalLength0 = lines.map { s->s.length }.reduce { a: Int, b:
Int -> a + b }

    // trimiterea unor functii catre Spark
    val totalLength1= lines.map(object : Function<String?, Int?> {
        override fun call(p0: String?): Int? {
            return p0?.length ?: 0
        }
    }).reduce(object : Function2<Int?, Int?, Int?> {
        override fun call(p0: Int?, p1: Int?): Int? {
            return (p0?.toInt() ?: 0) + (p1?.toInt() ?: 0)
        }
    })

    // sau daca scrierea functiilor inline sau a celor lambda este
greoaie, se pot utiliza clase
    val totalLength2 = lines.map(GetLength()).reduce(Sum())
    println(totalLength0)
    println(totalLength1)
    println(totalLength2)

    // variabila partajata de tip broadcast
    // trimiterea unui set de date ca input catre fiecare nod intr-o
maniera eficienta:
    val broadcastVar: Broadcast<List<Int>> =
sparkContext.broadcast(listOf(1, 2, 3))
    val totalLength3 = lines.map { s->s.length +
broadcastVar.value()[0] }.reduce { a: Int, b: Int -> a + b }
    println(totalLength3)

    //variabila partajata de tip acumulator

```

```

val accumulator = sparkContext.sc().longAccumulator()
sparkContext.parallelize(listOf(1, 2, 3, 4)).foreach { x ->
accumulator.add(x.toLong()) }
println(accumulator)

// oprirea contextului Spark
sparkContext.stop()
}

```

**Atenție:** înainte de execuția aplicației Spark RDD, se va crea fișierul `src/main/resources/data.txt` cu câteva linii de text oarecare

### Exemplul 2.1: Spark SQL - conectare la bază de date MySQL

În cazul în care nu aveți un server MySQL instalat și configurat local, puteți urma pașii de instalare disponibili la următorul URL: <https://linuxize.com/post/how-to-install-mysql-on-debian-10/>.

Pentru conectarea la o bază de date MySQL este necesară adăugarea următoarei dependențe în fișierul `pom.xml`:

```

<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
  <version>8.0.19</version>
</dependency>

```

Accesați consola MySQL astfel:

```
mysql -u root -p
```

Se va introduce parola pentru utilizatorul `root` al server-ului MySQL, configurată în pașii de instalare.

Creați un utilizator nou pentru Spark, din consola MySQL deschisă anterior:

```

CREATE USER 'spark'@'localhost' IDENTIFIED BY 'sparksql';
GRANT ALL PRIVILEGES ON *.* TO 'spark'@'localhost' WITH GRANT OPTION;
CREATE USER 'spark'@'%' IDENTIFIED BY 'sparksql';
GRANT ALL PRIVILEGES ON *.* TO 'spark'@'%' WITH GRANT OPTION;

```

Închideți consola MySQL `root` (folosind comanda `exit`) și conectați-vă cu noul utilizator „`spark`”:

```
mysql -u spark -p
```

Se va introduce parola „`sparksql`”.

Pentru crearea unei baze de date și a unei tabeli `Person`, precum și pentru popularea acesteia, se va executa următorul script în consola MySQL:

```

CREATE DATABASE IF NOT EXISTS sd_database;
USE sd_database;
CREATE TABLE IF NOT EXISTS Person (ID int PRIMARY KEY, LastName
varchar(255), FirstName varchar(255), Age int);
INSERT INTO Person(ID, LastName, FirstName, Age) VALUES(1, "Popescu",
"Ion", 20);
INSERT INTO Person(ID, LastName, FirstName, Age) VALUES(2, "Ionescu",
"Mariana", 20);
INSERT INTO Person(ID, LastName, FirstName, Age) VALUES(3, "Codreanu",

```

```
"Cristian", 20);
INSERT INTO Person(ID, LastName, FirstName, Age) VALUES (4, "Cantemir",
"Roxana", 21);
INSERT INTO Person(ID, LastName, FirstName, Age) VALUES (5, "Anghel",
"Vlad", 21);
INSERT INTO Person(ID, LastName, FirstName, Age) VALUES (6, "Bujor",
"Florina", 22);
COMMIT;
```

Observație: se poate pune scriptul într-un fișier `mysql_init.sql`, apoi dintr-un terminal deschis în directorul scriptului se pornește consola MySQL și se execută comanda:

```
source mysql_init.sql
```

### *Codul sursă*

Vezi comentariile cu explicații din cod.

```
package com.sd.laborator

import org.apache.spark.sql.SparkSession

fun main(args: Array<String>) {
    val spark = SparkSession.builder()
        .appName("Java Spark SQL example")
        .config("spark.master", "local[4]")
        .create()

    // Crearea unui DataFrame pe baza unei tabele "Person" stocata
    // intr-o baza de date MySQL
    //val url =
    "jdbc:mysql://yourIP:yourPort/databaseName?user=yourUsername&password=
    yourPassword&serverTimezone=UTC"
    val url =
    "jdbc:mysql://localhost:3306/sd_database?user=spark&password=sparksql&
    serverTimezone=UTC"
    val df = spark.sqlContext()
        .read()
        .format("jdbc")
        .option("url", url)
        .option("dbtable", "Person")
        .load()

    // Afisarea schemei DataFrame-ului
    df.printSchema()

    // Numararea persoanelor dupa varsta
    val countsByAge = df.groupBy("age").count()
    countsByAge.show()

    // Salvarea countsByAge in src/main/resources/sql_output in format
    JSON

    countsByAge.write().format("json").save("src/main/resources/sql_out-
    put")
}
```

Înlocuiești „**admin**” cu „**root**” și „**pass**” cu parola setată în pașii de configurare a server-ului MySQL.



*Exemplul 2.2: Spark SQL*

```

package com.sd.laborator

import org.apache.spark.api.java.JavaRDD
import org.apache.spark.api.java.function.MapFunction
import org.apache.spark.sql.*
import org.apache.spark.sql.functions.col
import java.io.Serializable

// clasa bean
class Person: Serializable {
    var name: String? = null
    var age = 0
}

fun main(args: Array<String>) {
    // configurarea si crearea sesiunii Spark SQL
    val sparkSession = SparkSession.builder()
        .appName("Java Spark SQL example")
        .config("spark.master", "local")
        .OrCreate

    // initializarea unui DataFrame prin citirea unui json
    val df: Dataset<Row> =
sparkSession.sqlContext().read().json("src/main/resources/people.json"
)

    // afisarea continutului din DataFrame la consola
    df.show()

    // Afisarea schemei DataFrame-ului intr-o forma arborescenta
    df.printSchema();

    // Selectarea coloanei nume si afisarea acesteia
    df.select("name").show();

    // Selectarea tuturor datelor si incrementarea varstei cu 1
    df.select(col("name"), col("age").plus(1)).show();

    // Selectarea persoanelor cu varsta > 21 ani
    df.filter(col("age").gt(21)).show();

    // Numararea persoanelor dupa varsta
    df.groupBy("age").count().show();

    // Inregistrarea unui DataFrame ca un SQL View temporar
    df.createOrReplaceTempView("people")

    // Utilizarea unei interogari SQL pentru a selecta datele
    val sqlDF: Dataset<Row> = sparkSession.sql("SELECT * FROM people")
    sqlDF.show()

    // Inregistrarea unui DataFrame ca un SQL View global temporar
    df.createGlobalTempView("people");

```

```

    // Un SQL View global temporar este legat de o baza de date a
    sistemului: `global_temp`
    sparkSession.sql("SELECT * FROM global_temp.people").show();

    // Un view global temporar este vizibil intre sesiuni
    sparkSession.newSession().sql("SELECT * FROM
    global_temp.people").show();

    // Crearea seturilor de date (Dataset)
    val person0 = Person() // instanta de clasa Bean
    person0.name = "Mihai"
    person0.age = 20
    val person1 = Person() // instanta de clasa Bean
    person1.name = "Ana"
    person1.age = 19

    // Este creat un codificator (Encoder) pentru bean-ul Java
    val personEncoder: Encoder<Person> =
    Encoders.bean(Person::class.java)
    // Se creeaza Dataset-ul de persoane
    val javaBeanDS: Dataset<Person> =
    sparkSession.createDataset(listOf(person0, person1), personEncoder)
    javaBeanDS.show()

    // Codificatoarele (Encoders) pentru tipurile primitive sunt
    furnizate de clasa Encoders
    val integerEncoder = Encoders.INT()
    val primitiveDS: Dataset<Int> =
    sparkSession.createDataset(listOf(1, 2, 3), integerEncoder)
    val transformedDS = primitiveDS.map(
        MapFunction { value: Int -> value + 1 } as MapFunction<Int,
    Int>,
        integerEncoder
    )
    transformedDS.collect() // [2, 3, 4]
    transformedDS.show() // afisarea listei transformate

    // DataFrame-urile pot fi convertite intr-un Dataset
    val path = "src/main/resources/people.json"
    val peopleDS: Dataset<Person> =
    sparkSession.read().json(path).as(personEncoder)
    peopleDS.show()

    // Interoperabilitatea cu RDD-uri
    // Crearea unui RDD de obiecte Person dintr-un fisier text
    val peopleRDD: JavaRDD<Person> = sparkSession.read()
        .textFile("src/main/resources/people.txt")
        .javaRDD()
        .map { line ->
            val parts: List<String> = line.split(",")
            val person = Person()
            person.name = parts[0]
            person.age = parts[1].trim { it <= ' ' }.toInt()
            person // return
        }

    // Aplicarea unei scheme pe un RDD de Java Bean-uri pentru a
    obtine DataFrame

```

```

val peopleDF: Dataset<Row> =
sparkSession.createDataFrame(peopleRDD, Person::class.java)
// Inregistrarea DataFrame-ului ca un view temporar
peopleDF.createOrReplaceTempView("people")

// Selectarea persoanelor intre 13 si 19 ani cu o interogare SQL
val teenagersDF: Dataset<Row> = sparkSession.sql("SELECT name FROM
people WHERE age BETWEEN 13 AND 19")

// Coloanele dintr-un Row din rezultat pot fi accesate dupa
indexul coloanei
val stringEncoder = Encoders.STRING()
val teenagerNamesByIndexDF = teenagersDF.map(
    MapFunction { row: Row -> "Name: " + row.getString(0) } as
MapFunction<Row, String>,
    stringEncoder
)
teenagerNamesByIndexDF.show()

// sau dupa numele coloanei
val teenagerNamesByFieldDF = teenagersDF.map(
    MapFunction { row: Row -> "Name: " + row.getAs("name") } ,
    stringEncoder
)
teenagerNamesByFieldDF.show()
}

```

**Înainte de a executa aplicația, adăugați următoarele fișiere în proiect:**

- **src/main/resources/people.json**

```

{"name": "Ionel"}
{"name": "Maria", "age": 23}
{"name": "Vasile", "age": 18}
{"name": "Ana", "age": 23}

```

- **src/main/resources/people.txt**

```

Ionel, 17
Maria, 23
Vasile, 18
Ana, 23

```

### *Exemplul 3: Integrarea Spark Streaming cu Kafka*

```

package com.sd.laborator

import org.apache.kafka.clients.consumer.ConsumerRecord
import org.apache.kafka.common.serialization.StringDeserializer
import org.apache.spark.SparkConf
import org.apache.spark.TaskContext
import org.apache.spark.api.java.JavaRDD
import org.apache.spark.api.java.JavaSparkContext
import org.apache.spark.streaming.Durations
import org.apache.spark.streaming.api.java.JavaInputDStream
import org.apache.spark.streaming.api.java.JavaStreamingContext
import org.apache.spark.streaming.kafka010.*
import scala.Tuple2

```

```

fun main() {
    // configurarea Kafka
    val kafkaParams = mutableMapOf<String, Any>(
        "bootstrap.servers" to "localhost:9092",
        "key.deserializer" to StringDeserializer::class.java,
        "value.deserializer" to StringDeserializer::class.java,
        "group.id" to "use_a_separate_group_id_for_each_stream",
        "auto.offset.reset" to "latest",
        "enable.auto.commit" to false
    )

    // configurarea Spark
    val sparkConf =
SparkConf().setMaster("local[4]").setAppName("KafkaIntegration")

    // initializarea contextului de streaming
    val streamingContext = JavaStreamingContext(sparkConf,
Durations.seconds(1))

    val topics = listOf("topicA", "topicB")

    // crearea unui flux de date direct (DirectStream)
    val stream: JavaInputDStream<ConsumerRecord<String, String>> =
KafkaUtils.createDirectStream(
        streamingContext,
        LocationStrategies.PreferConsistent(),
        ConsumerStrategies.Subscribe(topics, kafkaParams)
    )
    stream.mapToPair{record: ConsumerRecord<String, String> ->
Tuple2(record.key(), record.value()) }

    // crearea unui RDD (set de date imutabil distribuit)
    val offsetRanges =
        arrayOf( /* topicul, partitia, offset-ul de inceput, offset-ul
final */
            OffsetRange.create("test", 0, 0, 100),
            OffsetRange.create("test", 1, 0, 100)
        )
    val rdd: JavaRDD<ConsumerRecord<String, String>> =
KafkaUtils.createRDD(
        streamingContext.sparkContext(),
        kafkaParams,
        offsetRanges,
        LocationStrategies.PreferConsistent()
    )

    // obtinerea offset-urilor
    stream.foreachRDD { rdd ->
        val offsetRanges = (rdd.rdd() as
HasOffsetRanges).offsetRanges()
        rdd.foreachPartition { consumerRecords ->
            val o = offsetRanges.get(TaskContext.get().partitionId())
            println(
                o.topic() + " " + o.partition() + " " + o.fromOffset()
+ " " + o.untilOffset()
            )
        }
    }
}

```

```

    }

    // stocarea offset-urilor
    stream.foreachRDD { rdd ->
        val offsetRanges = (rdd.rdd() as HasOffsetRanges).offsetRanges()
        (stream.inputDStream() as
        CanCommitOffsets).commitAsync(offsetRanges)
    }

    streamingContext.start()
    streamingContext.awaitTerminationOrTimeout(1000)
}

```

## Aplicații și teme

### Aplicații de laborator

1. Consultând documentația oficială, reluați exemplul 1 și configurați Spark să fie executat local pe câte *core*-uri sunt disponibile și cu un număr de maxim 6 încercări de execuție a unui task (RDD).
2. Să se calculeze cu ajutorul Spark RDD, Spark SQL și Spark Streaming (deci 3 aplicații diferite) histograma caracterelor din alfabetul latin ([a-zA-Z]) dintr-un fișier text de tip *ebook*.

**Observație:** când se utilizează Spark Streaming, fișierele text trebuie create în timp real (în timpul execuției aplicației).

3. Să se implementeze un **Kafka Producer** în Python care să citească un fișier text, să extragă cuvintele și să le publice într-un topic. Să se creeze un direct stream folosind Spark Streaming și din histograma cuvintelor, să se returneze cele mai frecvente 15 cuvinte.

**Observație:** nu uitați să porniți server-ul Kafka (a se consulta laboratorul 10 pentru detalii).

### Teme pe acasă

- Să se implementeze un Kafka Producer în Python care să preia coordonatele mouse-ului timp de 10 secunde și să le publice într-un topic (mouse-ul va trebui „plimbat” pe diagonala ecranului). Pornind de la exemplul 3, să se preia fluxul de puncte utilizând Spark Streaming și să se realizeze o regresie liniară, calculând *covarianța*.

Pentru preluarea coordonatelor mouse-ului în Python, se poate utiliza scriptul de mai jos:

```

from pynput.mouse import Controller
import time

mouse = Controller()

if __name__ == "__main__":
    for _ in range(100):
        print(mouse.position)
        time.sleep(0.1)

```

Trebuie instalată dependența **pynput**:

```

python3 -m venv env
source env/bin/activate
pip3 install pynput==1.6.8

```