



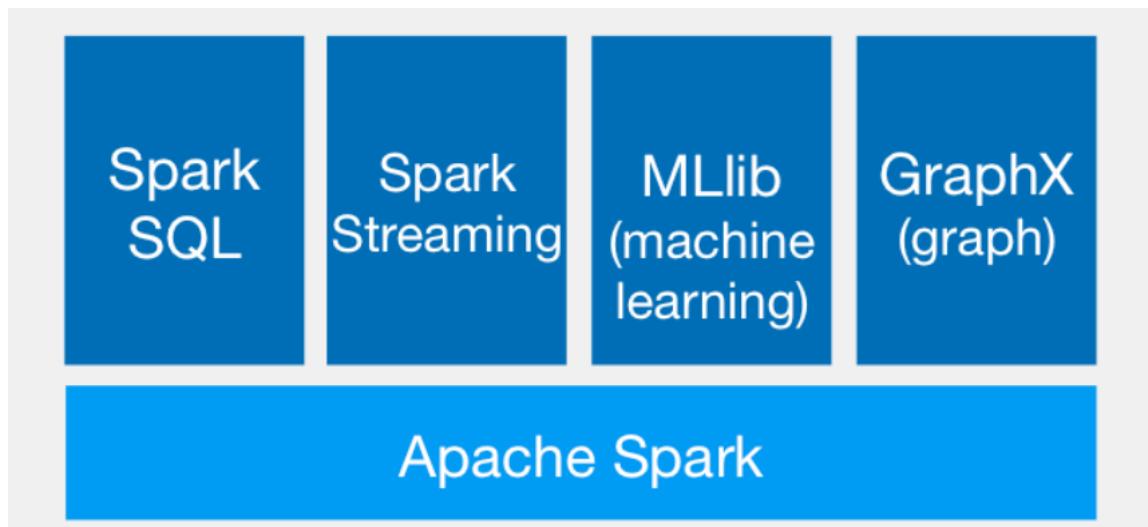
# **B I G D A T A**

CURS 5



**Apache Spark SQL**  
**Apache Spark Structured Streaming**  
**Apache Spark Streaming**

# Introducere



# SparkSession vs Spark/SQLContext

- În exemplele practice din cursul anterior am inițializat un obiect *SparkContext*, iar apoi (pe baza acestuia) un obiect *SQLContext*
- Putem avea un punct de intrare unificat (Spark  $\geq 2.0$ ): *SparkSession*
- *SparkSession* oferă un mod mai simplu (mai puține obiecte construite) de a interacționa cu diferite funcționalități Spark
  - Încapsulează Spark context, Hive context, SQL context
- Anterior versiunii 2.0 *SparkContext* a fost singurul punct de intrare pentru orice aplicație Spark, fiind folosit pentru a accesa restul funcționalităților / componentelor Spark.

# SparkSession vs Spark/SQLContext

- Cu ajutorul *SparkContext* putem crea doar RDD-uri
- Erau necesare contexte Spark specifice pentru alte interacțiuni cu Spark: *SQLContext*, *HiveContext*, pentru aplicațiile de *Streaming*.
- *SparkSession* este o combinație a acestor contexte diferite.
  - Intern, *SparkSession* crează un *SparkContext* pentru fiecare operație
  - Toate contextele menționate anterior pot fi accesate folosind obiectul *SparkSession*.
- Alt avantaj al *SparkSession*: în cazul mai multor utilizatori care accesează același *SparkContext*.

# Crearea unei sesiuni Spark

```
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName("Spark SQL Test") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
```

# Crearea unui *DataFrame*

- În exemplele din cursul 4 am creat *DataFrame*-uri pornind de la RDD-uri
- Alte metode: din sursele de date Spark (fișier *Parquet*, fișier JSON, tabel Hive, JDBC către alte BD, fișier avro, fișiere binare)

# Crearea unui *DataFrame*

```
# spark = obiectul SparkSession creat anterior  
df = spark.read.json("people.json")  
# afișarea DataFrame-ului la consolă  
df.show()
```

```
+---+---+  
| age | name |  
+---+---+  
| null | Michael |  
| 30 | Andy |  
| 19 | Justin |  
+---+---+
```

# Operații pe *DataFrame*-uri

- *DataFrame* = *DataSet* de (obiecte) *Row* în API-urile Java și Scala
- *DataFrame* API oferă un limbaj DSL (*Domain Specific Language*) pentru prelucrarea datelor structurate
- Operațiile pe *DataFrame*-uri -> transformări *untyped*
- Operațiile pe *DataSet*-uri -> transformări *typed* (*DataSet*-urile Java/Scala sunt puternic tipizate)

# Operații pe *DataFrame*-uri

- Accesarea unei coloane
  - În Python accesarea unei coloane se poate face:
    - Prin atribut: <data\_frame>.<atribut>
    - Prin indexare: <data\_frame>['<atribut>']
- Lista completă a operațiilor:  
<https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql.html#dataframe-apis>
- Funcții utile (prelucrare siruri de caractere, date calendaristice, operații aritmetice):  
<https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql.html#functions>

# Exemplu

```
# Afisarea schemei (format arbore)
df.printSchema()
# root
# |-- age: long (nullable = true)
# |-- name: string (nullable = true)

# Selectarea și afișarea coloanei
"name"
df.select("name").show()
# +-----+
# |    name|
# +-----+
# |Michael|
# |    Andy|
# | Justin|
# +-----+

# Selectarea tuturor persoanelor, cu
vârsta mărită cu 1
df.select(df['name'], df['age'] +
1).show()
# +-----+-----+
# |    name| (age + 1) |
# +-----+-----+
# |Michael|      null |
# |    Andy|        31 |
# | Justin|        20 |
# +-----+-----+
```

# Exemplu

```
# Selectarea persoanelor cu vârstă > 21
df.filter(df['age'] > 21).show()
# +---+-----+
# | age|name|
# +---+-----+
# | 30|Andy|
# +---+-----+  
  
# Numărarea persoanelor după vîrstă
df.groupBy("age").count().show()
# +---+-----+
# | age|count|
# +---+-----+
# | 19|     1|
# | null|    1|
# | 30|     1|
# +---+-----+'  
  
# +---+-----+
```

# Execuția programatică a cererilor SQL

- Funcția *sql* din *SparkSession* permite aplicațiilor să ruleze programatic cereri SQL și să returneze un rezultat de tip *DataFrame*.

```
# Înregistrează obiectul DataFrame ca view SQL temporara
df.createOrReplaceTempView("people")

sqlDF = spark.sql("SELECT * FROM people")
sqlDF.show()
# +-----+
# | age|    name|
# +-----+
# | null|Michael|
# |   30|    Andy|
# |   19| Justin|
```

# Vizualizare temporară globală

- Vizualizările temporare din Spark SQL sunt obiecte la nivel de sesiune – vor dispărea atunci când se încheie sesiunea care le-a creat.
- Vizualizare temporară globală – vizualizare temporară care este partajată între mai multe sesiuni, ce este menținută până când se încheie aplicația Spark.
- Este legată de o bază de date menținută de către sistem (*global\_temp*) și trebuie să calificăm numele vizualizării, atunci când o referim, cu numele acestei baze de date.

# Vizualizare temporară globală

```
# Înregistrarea DataFrame-ului ca vizualizare temporară globală
df.createGlobalTempView("people")

# Vizualizarea temporară globală se află în global_temp
spark.sql("SELECT * FROM global_temp.people").show()
# +-----+
# | age|    name|
# +----+----+
# | null|Michael|
# |   30|    Andy|
# |   19| Justin|
# +----+----+

# Vizualizarea temporară globală este disponibilă și în alte sesiuni
spark.newSession().sql("SELECT * FROM global_temp.people").show()
# +-----+
# | age|    name|
# +----+----+
# | null|Michael|
# |   30|    Andy|
# |   19| Justin|
# +----+----+
```

# Interoperabilitatea cu RDD

- Spark SQL suportă două metode diferite pentru convertirea RDD-urilor existente în *Dataset-uri*.
- Prima metodă utilizează reflexia pentru a deduce schema unui RDD ce conține tipuri specifice de obiecte
  - Această metodă înseamnă cod mai concis și funcționează bine atunci când cunoaștem deja schema.
- A doua metodă presupune utilizarea unei interfețe de programare ce permite construirea unei scheme și apoi aplicarea acesteia unui RDD existent.
  - Mai mult cod
  - Permite construirea de *Dataset-uri* atunci când coloanele și tipurile de date ale acestora nu sunt cunoscute până la runtime.

# Deducerea schemei utilizând reflexia

- Spark SQL poate converti un RDD de obiecte *Row* într-un *DataFrame*, deducând tipurile de date.
- Obiectele *Row* sunt construite prin transmiterea unei liste de perechi cheie/valoare drept *kwargs* clasei *Row*.
- Cheile acestei liste definesc numele coloanelor tabelului, iar tipurile sunt deduse eșantionând întregul *dataset*, similar inferenței asupra fișierelor JSON.

# Deducerea schemei utilizând reflexia

```
from pyspark.sql import Row

sc = spark.sparkContext

# Încarcă fișierul text și convertește fiecare linie într-un Row
lines = sc.textFile("people.txt")
parts = lines.map(lambda l: l.split(","))
people = parts.map(lambda p: Row(name=p[0], age=int(p[1])))

# Deduce schema și înregistrează DataFrame-ul ca tabel
schemaPeople = spark.createDataFrame(people)
schemaPeople.createOrReplaceTempView("people")

# Cererile SQL pot fi executate asupra DataFrame-uriloe care au fost înregistrate ca tabele.
teenagers = spark.sql("SELECT name FROM people WHERE age >= 13 AND age <= 19")

# Rezultatele cererilor sunt obiecte DataFrame.
# rdd returnează conținutul ca obiect :class:`pyspark.RDD` al :class:`Row`.
teenNames = teenagers.rdd.map(lambda p: "Name: " + p.name).collect()
for name in teenNames:
    print(name)
# Name: Justin
```

# Specificarea programatică a schemei

- Atunci când un dicționar de *kwargs* nu poate fi definit anterior, un *DataFrame* se poate crea programatic urmând 3 pași:
  1. Crearea unui RDD de tupluri sau a unei liste din RDD-ul original
  2. Crearea schemei reprezentate de un *StructType* corespunzător structurii tuplurilor sau listelor din RDD-ul construit la pasul anterior
  3. Aplicarea schemei asupra RDD-ului prin intermediul metodei *createDataFrame* din *SparkSession*.

# Specificarea programatică a schemei

```
# Import tipuri de date
from pyspark.sql.types import StringType, StructType, StructField

sc = spark.sparkContext

# Încărcarea unui fișier text și convertirea fiecărei linii în Row.
lines = sc.textFile("examples/src/main/resources/people.txt")
parts = lines.map(lambda l: l.split(","))
# Fiecare linie este convertită într-un tuplu
people = parts.map(lambda p: (p[0], p[1].strip()))

# Schema este encodată într-un string.
schemaString = "name age"

fields = [StructField(field_name, StringType(), True) for field_name in schemaString.split()]
schema = StructType(fields)

# Aplicarea schemei pe RDD.
schemaPeople = spark.createDataFrame(people, schema)

# Crearea unui view temporar pe baza DataFrame-ului
schemaPeople.createOrReplaceTempView("people") # +-----+
# | name| # +-----+
# | Michael| # +-----+
# | Andy| # +-----+
# | Justin| # +-----+
# +-----+ # +-----+
# SQL poate fi executat pe DataFrame-uri care au fost înregistrate ca tabele.
results = spark.sql("SELECT name FROM people")

results.show()
```

# Functii scalare

- Sunt funcții ce returnează câte o valoare (singulară) pentru fiecare *Row*
- Spark SQL suportă o diversitate de funcții scalare predefinite: <https://spark.apache.org/docs/latest/sql-ref-functions.html#scalar-functions>
- De asemenea, suportă funcțiile scalare definite de utilizator (UDF):  
<https://spark.apache.org/docs/latest/sql-ref-functions-udf-scalar.html>

# Exemplu UDF cu Python

- Demo în Jupyter Lab

# Functii agregat

- Sunt funcții ce returnează o singură valoare pentru un grup de linii (*Row*)
- Funcțiile agregat predefinite suportă tipurile de agregare obișnuite, precum *count*, *countDistinct*, *avg*, *max*, *min*
  - <https://spark.apache.org/docs/latest/sql-ref-functions-built-in.html#aggregate-functions>
- Utilizatorii pot crea și funcții agregat (UDAF – User Defined Aggregate Functions) în Scala și Java
  - <https://spark.apache.org/docs/latest/sql-ref-functions-udf-aggregate.html>

# Menținerea datelor în memorie (*caching*)

- Spark SQL poate păstra tabelele în *cache* folosind un format columnar. Apelăm:
  - `spark.catalog.cacheTable("tableName")` sau
  - `dataFrame.cache()`
- Apoi, Spark SQL va parcurge doar coloanele necesare și va optimiza compresia astfel încât utilizarea memoriei și a presiunii asupra GC (GC Pressure – Garbage Collector Pressure) să fie minime.
- Ștergerea tabelului din memorie:
  - `spark.catalog.uncacheTable("tableName")`

# Menținerea datelor în memorie (*caching*)

- Configurarea *caching*-ului *in-memory* se poate realiza cu ajutorul metodei *setConf* asupra *SparkSession* sau prin execuția de comenzi *SET cheie=valoare* folosind SQL
- *spark.sql.inMemoryColumnarStorage.compressed* (implicit *true*) – Spark SQL va selecta automat un codec pentru compresia fiecărei coloane, pe baza statisticilor datelor
- *spark.sql.inMemoryColumnarStorage.batchSize* (implicit 10000) – controlează dimensiunea *batch*-urilor pentru *caching*-ul columnar. Cu cât dimensiunea este mai mare, cu atât este îmbunătățită utilizarea memoriei și compresia, dar apare riscul de erori *OutOfMemory* la *caching*-ul datelor.

# Strategii pentru operația de *join*

- Hint-urile pentru strategia de join:
  - *BROADCAST*
  - *MERGE, SHUFFLE\_HASH* și *SHUFFLE\_REPLICATE\_NL* ( $\geq 3.0$ )
- Aceste *hint*-uri instruiesc Spark să folosească o anumită strategie pe fiecare relație specificată, atunci când are loc operația de *join* a acesteia cu o altă relație.
- *spark.sql.autoBroadcastJoinThreshold* – dimensiunea maximă a unui tabel ce va fi transmis tuturor nodurilor executor atunci când are loc *join*-ul.
- Hint *BROADCAST* pe *table1 => table1* va avea prioritate chiar dacă dimensiunea lui *table1* este mai mare decât valoarea parametrului *spark.sql.autoBroadcastJoinThreshold*
- Prioritate: *broadcast > merge > shuffle\_hash > shuffle\_hash\_nl*

```
spark.table("src").join(spark.table("records")).hint("broadcast"),  
"key").show()
```

# Strategii pentru operația de *join*

- Broadcast *join* – efectuează join-ul pe 2 relații prin:
  - *broadcast*-ul relației mai mici în toți executorii
  - Evaluarea criteriului de join cu partiiile celeilalte relații aflate pe executorul respectiv

# Pandas cu Apache Arrow

- Apache Arrow este un format de date columnar, *in-memory*, utilizat de către Spark pentru a face transferul eficient al datelor între procesele JVM și Python.
- Benefic utilizatorilor Python care lucrează cu date Pandas/NumPy.
- Utilizarea nu este automată și necesită modificări în configurare sau cod.
- Pandas este o librărie open-source, cu licență BSD, ce furnizează performanță, structuri de date ușor de folosit și tool-uri de analiză a datelor pentru limbajul Python.
  - PySpark suportă Pandas

# PySpark și Pandas

- PySpark poate converti date între (tipul) *DataFrame* din PySpark și *DataFrame* din Pandas
  - `pdf = df.toPandas()`
  - `df = spark.createDataFrame(pdf)`
- Putem folosi *Arrow* ca format intermediar, prin configurarea `spark.sql.execution.arrow.pyspark.enabled = true` (implicit este *false*)

# Pandas cu Apache Arrow

- Arrow permite optimizarea conversiei între cele două tipuri *DataFrame*.
- Aceste optimizări pot să nu fie luate în considerare (și să aibă loc optimizări non-Arrow) dacă apar erori.
  - Parametrul care controlează acest lucru este `spark.sql.execution.arrow.pyspark.fallback.enabled`

# Pandas cu Apache Arrow

```
import numpy as np
import pandas as pd

# Activarea transferurilor de date pe coloane bazate pe Arrow
spark.conf.set("spark.sql.execution.arrow.pyspark.enabled", "true")

# Generarea unui DataFrame Pandas
pdf = pd.DataFrame(np.random.rand(100, 3))

# Crearea unui DataFrame Spark din DataFrame-ul Pandas folosind Arrow
df = spark.createDataFrame(pdf)

# Convertirea DataFrame-ului Spark înapoi la Pandas DataFrame folosind
# Arrow
result_pdf = df.select("*").toPandas()
```

# Pandas cu Apache Arrow

- Prin apelul *DataFrame.toPandas()* se colectează toate înregistrările din *DataFrame* în *Driver Program* => ar trebui să se efectueze pe o submulțime a datelor
- Nu toate tipurile de date Spark sunt suportate și pot apărea erori dacă o coloană are un tip care nu este suportat
- Dacă apare o eroare în cadrul operației *SparkSession.createDataFrame()*, atunci Spark va crea *DataFrame-ul* fără *Arrow*

# Functii definite de utilizator (UDF) în Pandas

- Pandas UDFs = Vectorized UDFs
- Sunt funcții definite de utilizator ce sunt executate de către Spark folosind *Arrow* pentru transferul datelor și Pandas pentru a lucra cu datele, lucru ce permite operațiile vectorizate.
- Un UDF Pandas se definește folosind decoratorul `pandas_udf()` și se comportă ca orice funcție PySpark
- Python  $\geq 3.6$  permite utilizarea *hint-urilor* de tipuri

# Functii definite de utilizator (UDF) în Pandas

```
import pandas as pd

from pyspark.sql.functions import pandas_udf

@pandas_udf("coll string, col2 long")
def func(s1: pd.Series, s2: pd.Series, s3: pd.DataFrame) -> pd.DataFrame:
    s3['col2'] = s1 + s2.str.len()
    return s3

# Create a Spark DataFrame that has three columns including a struct column.
df = spark.createDataFrame(
    [[1, "a string", ("a nested string",)],],
    "long_col long, string_col string, struct_col struct<col1:string>")

df.printSchema()
# root
# |-- long_column: long (nullable = true)
# |-- string_column: string (nullable = true)
# |-- struct_column: struct (nullable = true)
# |   |-- col1: string (nullable = true)

df.select(func("long_col", "string_col", "struct_col")).printSchema()
# |-- func(long_col, string_col, struct_col): struct (nullable = true)
# |   |-- col1: string (nullable = true)
# |   |-- col2: long (nullable = true)
```

# Functii definite de utilizator (UDF) în Pandas

- Hint-ul de tip utilizează *pandas.Series* în toate cazurile, dar există un caz în care se utilizează hint-ul *pandas.DataFrame*: atunci când coloana (de intrare sau ieșire) este de tip *StructType*
- *Hint-uri* de tip suportate:
  - *pandas.Series, ... -> pandas.Series*
  - *Iterator[pandas.Series] -> Iterator[pandas.Series]*
  - *Iterator[Tuple[pandas.Series, ...]] -> Iterator[pandas.Series]*
  - *pandas.Series, ... -> Any* (orice tip scalar)
  - + varianta *pandas.DataFrame*

# Apache Structured Streaming

# Introducere

- *Structured Streaming* este un motor de procesare a *stream-urilor* de date, scalabil și rezistent la defecte, construit peste motorul Spark SQL
- Permite exprimarea calculului pe *stream-uri* în mod similar calculului *batch* pe date statice.
- Motorul Spark SQL va rula codul incremental și continuu și va actualiza rezultatul final, pe măsură ce datele de *streaming* continuă să sosească.
- API-ul Dataset/DataFrame poate fi utilizat (în Scala, Java, Python, R) pentru a exprima agregări, ferestre eveniment-timp, join-uri stream-to-batch etc.
- Execuția se realizează pe același motor Spark SQL optimizat.

# Introducere

- *Structured Streaming* furnizează procesare de stream-uri rapidă, scalabilă, rezistentă la defecte, *end-to-end exactly-once*, fără ca utilizatorul să prevadă acțiuni specifice.
- Intern, cererile Structured Streaming sunt procesate utilizând un motor de procesare *micro-batch*, ce procesează stream-urile de date ca pe o serie de job-uri batch mai mici, obținând latențe end-to-end de ordinul a 100 ms și garanția rezistențelor la defecte *exactly-once*.
- Din versiunea Spark 2.3 a fost introdus un nou mod de procesare (cu latență scăzută) numit *Continuous Processing* – obține latențe end-to-end de ordinul a 1ms cu garanție de tip *at-least-once*.

# Introducere

- Fără a modifica operațiile Dataset/DataFrame din cereri, modul va putea fi ales pe baza cerințelor aplicației.
- Modul de procesare implicit este *micro-batch*.

# Exemplu

- O aplicație simplă: primim evenimente (date) de forma (phone\_id, time, action) de la o aplicație mobilă și dorim să numărăm câte acțiuni de fiecare tip au loc în fiecare oră, iar apoi să stocăm rezultatul în MySQL
- Dacă aplicația era executată ca *batch job* și aveam un tabel cu toate evenimentele de intrare, am fi putut exprima sub forma unui query SQL:

```
SELECT action, WINDOW(time, "1 hour"), COUNT(*)  
FROM events  
GROUP BY action, WINDOW(time, "1 hour")
```

# Modelul Structured Streaming

- Sistemul garantează că la orice moment, rezultatul aplicației este echivalent cu execuția unui job de tip *batch* pe un prefix al datelor
- De exemplu: rezultatul (tabelul MySQL) va fi mereu echivalent cu luarea unui prefix al *stream*-ului actualizat al fiecărui telefon și rularea cererii SQL de mai sus
- Garanția integrității prefixului asigură că tabelele de ieșire vor fi întotdeauna consistente cu liniile din prefixul datelor.

# Modelul Structured Streaming

- Nu există pericolul datelor *out-of-order*
- API-ul este simplu de folosit: de fapt, este vorba tot despre DataFrame API.
- Utilizatorii descriu cererea, locația intrării și a ieșirii și (optional) alte detalii.
- Sistemul rulează apoi cererea incremental, menținând starea necesară pentru revenirea în urma unor potențiale căderi, menține rezultatele consistente în storage-ul extern etc.

# Modelul Structured Streaming

- Codul principal al aplicației de monitorizare este:

```
// Citire continua de la o sursa s3  
val inputDF = spark.readStream.json("s3://logs")  
  
// Operatii folosind DataFrame API si scriere in  
MySQL  
inputDF.groupBy($"action", window($"time", "1  
hour")).count()  
    .writeStream.format("jdbc")  
    .start("jdbc:mysql://...")
```

# Modelul Structured Streaming

- Codul este aproape identic cu versiune *batch* (s-au schimbat doar „read” și „write”):

```
// Citire date o singura data de la o locatie s3
val inputDF = spark.read.json("s3://logs")

// Operatii folosind DataFrame API si scriere in MySQL
inputDF.groupBy($"action", window($"time", "1
hour")).count()
    .writeStream.format("jdbc")
    .save("jdbc:mysql//...")
```

# Apache Spark Streaming

# Introducere

- Streaming = Date generate continuu dintr-una sau mai multe surse
- În general, sursele trimit datele simultan
- Datele ajung în pachete mici (de ordinul KB), succesiv

# Introducere

- O mulțime de aplicații folosesc date actualizate continuu
- Exemple:
  - Senzori din vehicule, echipament industrial, mașini ce trimit date în *streaming* pentru măsurarea performanțelor
  - Site web ce urmărește datele de geo-locație de pe telefoanele clienților, astfel încât site-ul să poată face recomandări (de exemplu, ce obiective pot fi vizitate)
  - Companie de energie solară ce monitorizează performanța panourilor prin *streaming*
  - Companie de *gaming online* ce colectează date de *streaming* despre interacțiunile dintre jucător și joc

# Tool-uri de streaming

Storm

Flink

Kinesis

Samza

Kafka

Apache  
Spark

TCP  
Socket



# Apache Spark Streaming

- Spark are un scop general și este folosit pe scară largă
- Spark se poate conecta cu multe dintre *tool*-urile de *streaming* existente
- Este rezistent la defecte

# Apache Spark Streaming

- Apache Spark Streaming este un sistem de procesare de *streaming* rezistent la defecte, ce suportă atât date de tip *batch* cât și date de *streaming*
- Este o extensie a nucleului Spark API ce permite procesarea în timp real a datelor din diferite surse
- Datele procesate pot fi trimise în sisteme de fișiere, baze de date sau *dashboard-uri live*

# Apache Spark Streaming



# Apache Spark Streaming

- Abstractizarea sa principală este *Discretized Stream (DStream)* – reprezintă un stream de date împărțit în *batch-uri* mai mici
- *DStream* – construit peste RDD (abstractizarea datelor din nucleul Spark)
- Permite Spark Streaming să se integreze cu ușurință cu orice altă componentă Spark (SQL, MLlib)

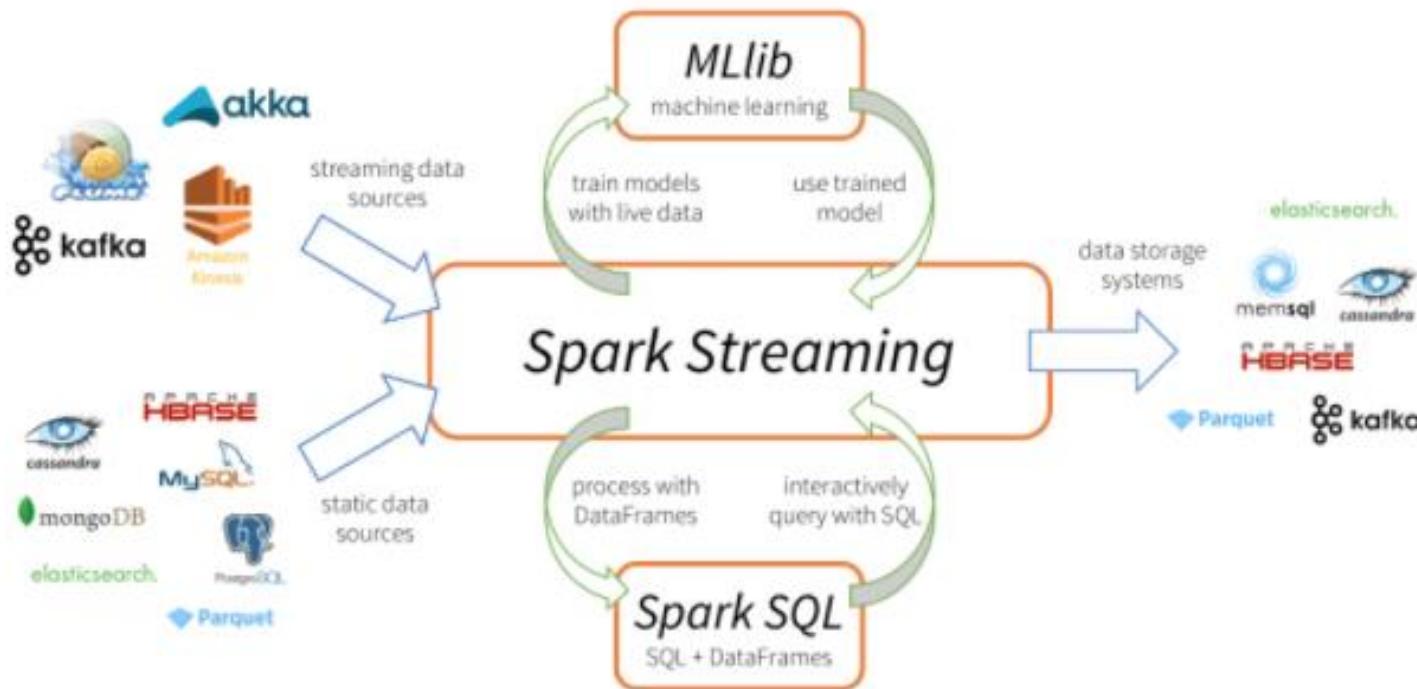
# Apache Spark Streaming

- Este diferit de alte sisteme care fie au un motor de procesare proiectat doar pentru *streaming*, fie au API-uri similare pentru *batch* și *streaming*, dar compilează intern la alte engine-uri.
- Motorul de execuție unic al Spark și modelul unificat de programare pentru *batch* și *streaming* conduc la beneficii unice față de sistemele tradiționale de *streaming*.

# Aspecte majore

- Recuperare rapidă în urma căderilor
- *Load balancing* și utilizare a resurselor mai bune
- Combină datele de *streaming* cu seturile de date statice și cererile interactive
- Integrare nativă cu librării avansate de procesare (SQL, ML, procesare de grafuri)

# Aspecte majore



- Această unificare a funcționalităților separate de procesare a datelor este motivul cheie din spatele adoptării rapide a Spark Streaming.
- Este mai simplu pentru dezvoltatori să folosească un singur *framework* care să întrunească toate funcționalitățile de procesare.

# Cum funcționează?

- Spark Streaming primește date de intrare live și le împarte în *batch*-uri
- Acestea sunt procesate de către motorul Spark pentru a rezulta *stream*-ul final (alcătuit, de asemenea, din batch-uri)



# Abstractizare

- *DStream (Discretized Stream)* – *stream* continuu de date
- Un DStream poate fi creat fie din *stream*-uri de date de intrare din surse precum cele amintite anterior (Kafka, Kinesis etc.) sau aplicând operații pe alte DStream-uri.
- Intern, un *DStream* este reprezentat ca o secvență de RDD-uri

# Programare cu Spark Streaming

- Programele Spark Streaming pot fi scrise în Scala, Java sau Python (Spark  $\geq 1.2$ )

# Bibliografie

1. Apache Spark documentation (<https://spark.apache.org/docs/latest>).
2. Apache Spark SQL Programming Guide (<http://spark.apache.org/docs/latest/sql-programming-guide.html>)
3. <https://docs.databricks.com/spark/latest/spark-sql/udf-python.html>
4. [https://spark.apache.org/docs/latest/api/python/user\\_guide/arrow\\_pandas.html](https://spark.apache.org/docs/latest/api/python/user_guide/arrow_pandas.html)
5. <https://www.slideshare.net/ueshin/apache-arrow-and-pandas-udf-on-apache-spark>
6. <https://spark.apache.org/docs/latest/sql-ref-syntax-qry-select-window.html>
7. <https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html>
8. <https://databricks.com/blog/2016/07/28/structured-streaming-in-apache-spark.html>
9. <https://spark.apache.org/docs/latest/streaming-programming-guide.html>