BIG DATA

CURS 6

Apache Spark Structured Streaming
Apache Spark Streaming

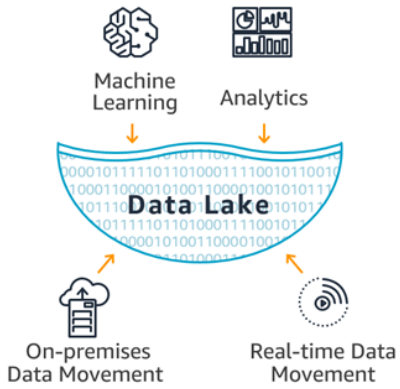
Data Pipeline

- *Data Pipeline* – reprezintă modul în care datele ajung din punctul A în punctul B; de la colectare până la rafinare, de la stocare până la analiză.
- Acoperă întregul proces de deplasare a datelor, de la locul în care sunt colectate (diverse dispozitive), locul și modul în care sunt mutate (prin *streaming* sau *batch processing*) și unde ajung (într-o aplicație sau un *data lake*).

Data Pipeline

What is a data lake?

A data lake is a centralized repository that allows you to store all your structured and unstructured data at any scale. You can store your data as-is, without having to first structure the data, and run different types of analytics—from dashboards and visualizations to big data processing, real-time analytics, and machine learning to guide better decisions.



[https://aws.amazon.com/big-data/datalakes-and-analytics/what-is-a-data-lake/]

Data Pipeline

[https://www.bmc.com/blogs/data-pipeline/]

Data Pipeline

- Un *Data Pipeline* trebuie să conducă datele către destinația lor, permițând fluxului de *business* să se desfășoare corect
- Dacă *Data Pipeline* nu funcționează, pot să nu fie generate rapoarte, comportamentul utilizatorilor să rămână neprocesat etc.
- Un *Data Pipeline* bun conduce datele de la sursă către destinație într-o manieră punctuală și sigură

Procesarea datelor: *stream* vs *batch*?

- *Streaming* și *batching* – cele mai comune modalități de procesare a datelor
- Datele de tip *streaming* sunt mutate de la punctul A în punctul B aproape în timp real. Este o formă de programare reactivă. *Stream*-ul de date este declanșat de un eveniment utilizator specific.
- La o postare pe Twitter – *tweet*-ul este parte dintr-un *data stream* care ia informația utilizatorului și o deplasează într-o zonă de „acces global” astfel încât ceilalți utilizatori să o poată vedea.
- Când se face o verificare a corectitudinii postărilor, are loc procesarea *tweet*-ului ca date în cadrul unui stream, combinat cu un micro-serviciu care permite analiza.

Procesarea datelor: *stream* vs *batch*?

- *Batch processing* – benefic pentru procesarea volumelor mari de date. Destinația poate aștepta un interval de timp (considerabil – de ordinul zile, săptămâni) astfel încât datele să poată fi deplasate la momente planificate.
- Exemplu de date *batch*: rapoarte de final de trimestru – date care nu sunt necesare imediat. Datele folosite pentru analiză, precum raportările care sunt mai puțin frecvente, pot fi de tip *batch*.

Transformări asupra datelor

- Datele din *pipeline* nu trebuie să fie neapărat transformate
- Dacă apar transformări, acestea vor face parte din *pipeline*
- Transformările pot fi de tipuri variate, de la transformări foarte simple până la transformări complexe
- Exemple: conversie de documente .doc în text brut, pentru a fi stocate uniform într-un *data lake*, modificarea unui tip de date (de la integer la string), clasificarea datelor de tip imagine etc.

Destinația datelor

- Poate influența dacă se aplică procesarea *stream* sau *batch*
- Destinația este mediul în care datele vor fi prezentate
- Pot fi foloste în scop personal, procesate pentru recunoașterea facială, compilate pentru raportări, pregătite pentru a antrena un model ML etc.
- Prezentate în mod cât mai lizibil, cu ajutorul tehnicilor de vizualizare a datelor

Exemple

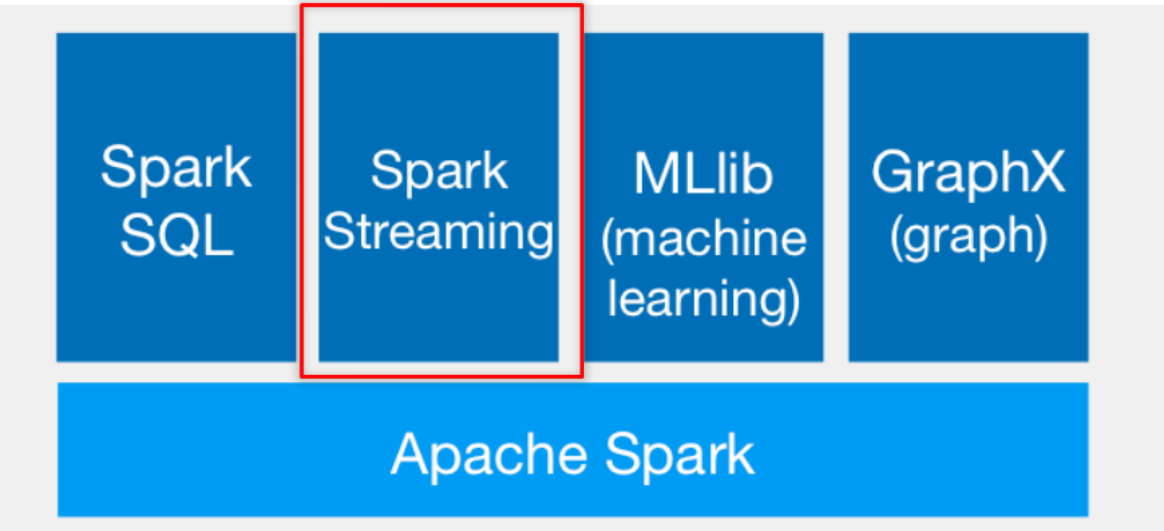
Exemple de cazuri ce necesită *data pipelines*:

- Stocarea cantităţilor mari de date
- Obţinerea de date din surse multiple
- Stocarea datelor în *cloud*
- Necesitatea operaţiilor rapide pentru analiză şi raportare.

Versionarea datelor

- O parte importantă a unui *data pipeline*
- *Workflow*-ul CI/CD DevOps necesită uneori revenirea la versiuni anterioare, atunci când una mai nouă eșuează, în vreme ce versiunea anterioară funcționează. Același concept se aplică pe datele organizațiilor.
- Alți indicatori de performanță (KPI – *Key Performance Indicator*):
  - Versionare
  - Latență
  - Scalabilitate
  - Interogare
  - Monitorizare
  - Testare

Streaming în Apache Spark



Streaming în Apache Spark

- *Structured Streaming* – componentă peste Spark SQL
- *Spark Streaming* – componentă separată

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*.

Introducere

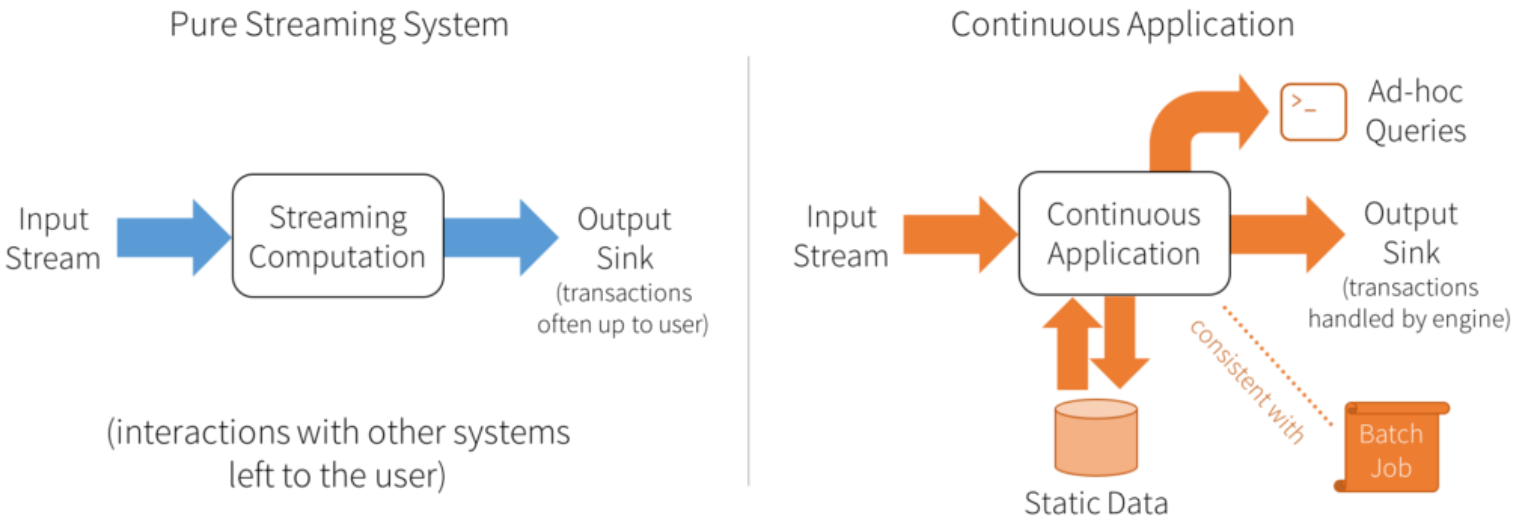
- *Structured Streaming* permite dezvoltarea de aplicații continue (*continuous applications*)
- Facilitează construirea aplicatiilor de *streaming end-to-end*, care se integrează cu nivelul de *storage*, sistemele de servire (în care sunt furnizate rezultatele) și *job*-urile de tip *batch* în mod consistent și rezistent la defecte
- Motoarele de *streaming* se ocupă, în general, doar de efectuarea operațiilor (calculelor) pe *stream*-uri
- În majoritatea cazurilor, procesarea *stream*-urilor face parte dintr-o aplicație mai mare – *Continuous Application*

Introducere

Exemple:

- ETL (Extract – Transform – Load) – deplasarea și transformarea continuă a datelor de la un sistem de stocare către altul (de exemplu, log-uri JSON în tabel Apache Hive) – o astfel de aplicație necesită o interacțiune atentă cu ambele sisteme de stocare pentru a asigura că datele nu sunt pierdute sau duplicate.
- Online machine learning – aplicații continue ce combină seturi de date statice, de dimensiuni mari, cu date real-time și cu servirea de predicții live.

Introducere



Ce se întâmplă, de obicei, în motoarele de streaming vs. ce ar fi necesar [https://databricks.com/blog/2016/07/28/continuous-applications-evolving-streaming-in-apache-spark-2-0.html]

Exemplu

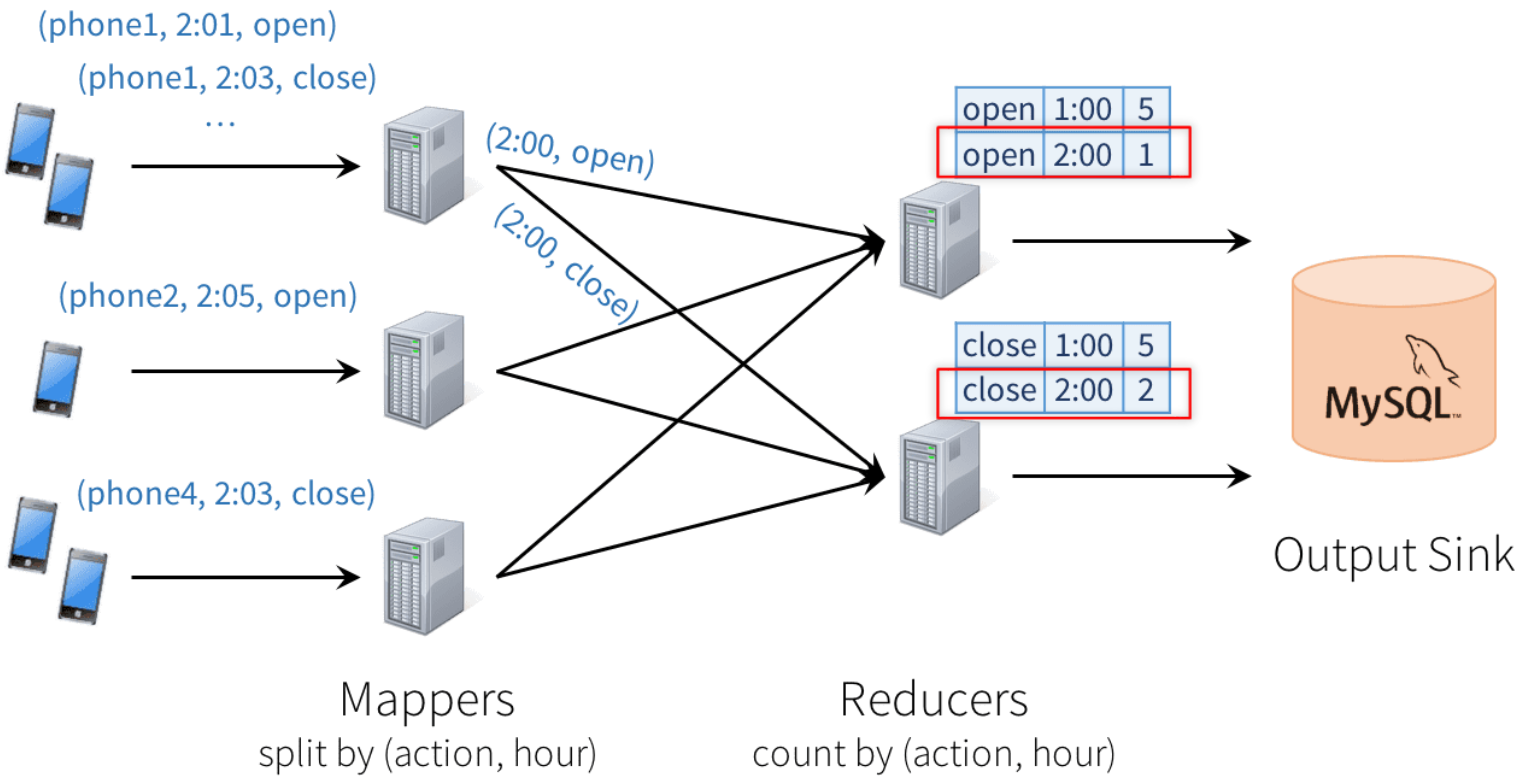
- O aplicație simplă: primim evenimente (date) de forma (phone_id, time, action) de la o aplicație mobilă și dorim să numărăm câte acțiuni de fiecare tip au loc în fiecare oră, iar apoi să stocăm rezultatul într-o bază de date (MySQL)
- Dacă aplicația era executată ca *batch* job și aveam un tabel cu 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")
```

Procesarea distribuită a *stream*-urilor

- De ce nu este suficientă lansarea unor servere care își transmit date între ele?
- Procesarea distribuită a *stream*-urilor prezintă unele dificultăți pe care nu le regăsim în procesările mai simple (precum cele de tip *batch job*).
- Exemplul anterior: „dacă aplicația era executată ca *batch job*...”
- Scenariu: Într-un motor de *streaming* distribuit, putem defini noduri care să proceseze datele după modelul *map-reduce*:

Procesarea distribuită a *stream-urilor*



[https://databricks.com/wp-content/uploads/2016/07/image00-2.png]

Procesarea distribuită a *stream*-urilor

- Fiecare nod de pe primul nivel:
  - citește o partiție a datelor de intrare (*stream*-ul asociat unei mulțimi de telefoane)
  - efectuează un *hash* pe evenimente după (action, hour)
  - trimite valorile obținute în urma acestei operații unui nod *reducer*, care numără evenimentele din acel grup și actualizează periodic baza de date MySQL
- Care sunt problemele acestui design?

Procesarea distribuită a *stream*-urilor

Probleme motor de streaming distribuit

1. **Consistenţa** – pot fi procesate înregistrări într-o parte a sistemului înainte de a fi procesate în alta => rezultate fără sens. Presupunem că aplicaţia trimite evenimentele *open* şi *close* atunci când utilizatorul deschide/închide aplicaţia. Dacă nodul *reducer* responsabil pentru *open* este mai lent decât cel pentru *close*, ar putea rezulta (în baza de date MySQL) un număr total de *close* mai mare decât numarul total de *open*
2. **Rezistenţa la defecte** – dacă nodurile *mapper* sau *reducer* eşuează: un *reducer* nu trebuie să numere o acţiune de 2 ori în baza de date, dar trebuie să poată solicita nodurilor *mapper* date mai vechi. Problema este dificilă; în multe motoare menţinerea unui rezultat consistent în *storage*-ul extern rămâne în sarcina utilizatorului.
3. **Date out-of-order** – în realitate, datele din surse diferite pot deveni *out-of-order*. De exemplu, un telefon poate încărca datele mai tîrziu, dacă nu a avut conexiune. Scrierea operatorilor *reducer* pe baza faptului că datele sosesc în ordinea câmpurilor dată-timp nu va funcţiona. Aceşti operatori trebuie să prevadă primirea de date out-of-order şi actualizarea corespunzătoare în baza de date.

Modelul Structured Streaming

-> În multe dintre sistemele de *streaming*, aceste aspecte revin (parţial sau integral) utilizatorului.

-> Problemele legate de modul în care aplicaţia interacţionează cu lumea reală – raţionament dificil => dificil de obţinut o semantică corespunzătoare

- Modelul Structured Streaming – Abordare a problemelor de semantică expuse anterior
- Sistemul garantează că la orice moment, rezultatul aplicaţiei este echivalent cu execuţia unui job de tip *batch* pe un prefix al datelor
- De exemplu: rezultatul (tabelul MySQL) va fi mereu echivalent cu considerarea unui prefix al *stream*-ului fiecărui telefon și rularea cererii SQL de mai sus

Modelul Structured Streaming

Garanţia integrităţii prefixului rezolvă cele 3 probleme identificate anterior:

1. Asigură că tabelele de ieşire vor fi întotdeauna consistente cu liniile din prefixul datelor.
2. Rezistenţa la defecte este asigurată de Structured Streaming, inclusiv în interacţiunile cu *sink*-urile externe.
3. Nu există pericolul datelor *out-of-order*

- API-ul este simplu de folosit: de fapt, este vorba tot despre DataFrame şi Dataset API.
- Utilizatorii descriu cererea, locaţia intrării şi a ieşirii şi (opţional) alte detalii.
- Sistemul rulează apoi cererea incremental, menţinând starea necesară pentru revenirea în urma unor potenţiale căderi, menţine rezultatele consistente în *storage*-ul extern etc.

Modelul Structured Streaming

- Codul principal al aplicației de monitorizare este de forma:

```
// 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 versiunea *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//…")
```

Agregări pe ferestre eveniment-timp

- Aplicațiile de *streaming* pot necesita calcule pe diferite „ferestre” de date, inclusiv ferestre care se suprapun (de exemplu, o fereastră de 1 oră care avansează câte 5 minute) sau care sunt disjuncte (de exemplu, pentru fiecare oră).
- În Structured Streaming, calculul pe ferestre este reprezentat printr-un *groupBy*
- Fiecare eveniment de intrare poate fi mapat într-una sau mai multe ferestre și conduce la actualizarea uneia sau mai multor linii din tabelul rezultat.
- Ferestrele pot fi specificate folosind functia *Window* din DataFrame.
- *Window* – face parte din pachetul pyspark.sql.functions

Agregari pe ferestre eveniment-timp

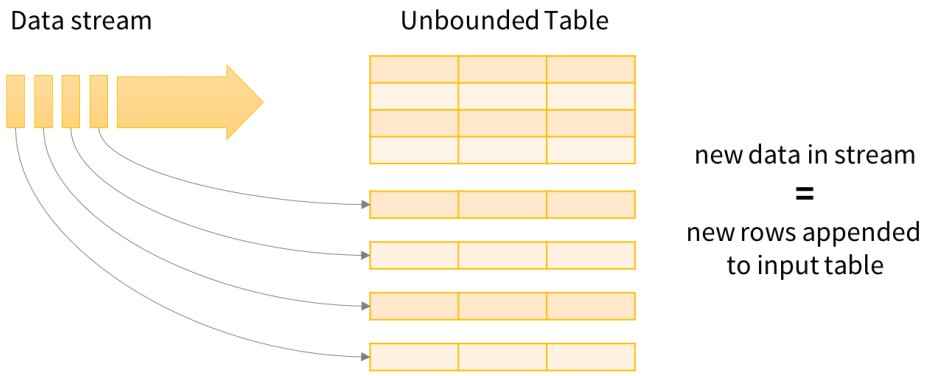
- Putem modifica job-ul anterior de monitorizare astfel încât să numere acţiunile pe ferestre de tip *sliding* astfel:

```
inputDF.groupBy($"action", window($"time", "1 hour", "5 minutes")).count()
```

- În prima variantă rezultatele sunt de forma (oră, acţiune, număr), iar în cea de-a doua (fereastră, acţiune, număr).
- Dacă o înregistrare ajunge târziu, atunci actualizăm (în MySQL) toate ferestrele corespunzătoare.
- Spre deosebire de alte sisteme, agregarea pe ferestre nu este un operator special pentru operaţiile de tip *streaming*. Acelaşi cod pare fi executat într-un job de tip *batch* pentru a grupa datele în acelaşi mod.
- https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#window-operations-on-event-time

Detaliile modelului Structured Streaming

- Conceptual, modelul Structured Streaming tratează toate datele pe care le primește ca pe un tabel de intrare nelimitat. Fiecare nou *item* din *stream* este similar unei linii adăugate în tabelul de intrare.
- Nu vor fi reținute toate datele de intrare, dar rezultatele vor fi echivalente cu existența întregului tabel și execuția unui job *batch*.



Data stream as an unbounded Input Table

Detaliile modelului Structured Streaming

- Dezvoltatorul definește o cerere pe acest tabel de intrare, ca și cum ar fi un tabel static
- Cererea determină un tabel rezultat ce va scris la un *sink* de ieșire
- Spark convertește automat cererea de tip *batch* într-un plan de execuție de tip *streaming*
- Acest procedeu se numește incrementalizare = Spark determină ce stare trebuie menținută pentru a actualiza rezultatul de fiecare dată când apare o înregistrare
- Dezvoltatorii specifică *trigger*-i pentru a controla cînd trebuie actualizate rezultatele. La declanșarea unui *trigger* Spark verifică datele noi (liniile noi din tabelul de intrare) și actualizează incremental rezultatul.

Structured Streaming API

- Structured Streaming este integrat în API-urile Dataset și DataFrame
- În majoritatea cazurilor este necesar doar să adăugăm câteva apeluri de metode pentru a executa operații pe *stream*-uri.
- Adaugă operatori noi pentru agregările pe ferestre și pentru setarea parametrilor modelului de execuție.
- Un *stream* – un DataFrame cu proprietatea *isStreaming* = true
- https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html#api-using-datasets-and-dataframes

Demo Structured Streaming

- Databricks (https://databricks.com/)
- https://community.cloud.databricks.com/
  - Creare cont
  - Creare cluster
  - Import notebook și atasare în cluster-ul creat anterior:
- https://docs.databricks.com/_static/notebooks/structured-streaming-python.html
  - Demo

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 Spark Streaming 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)

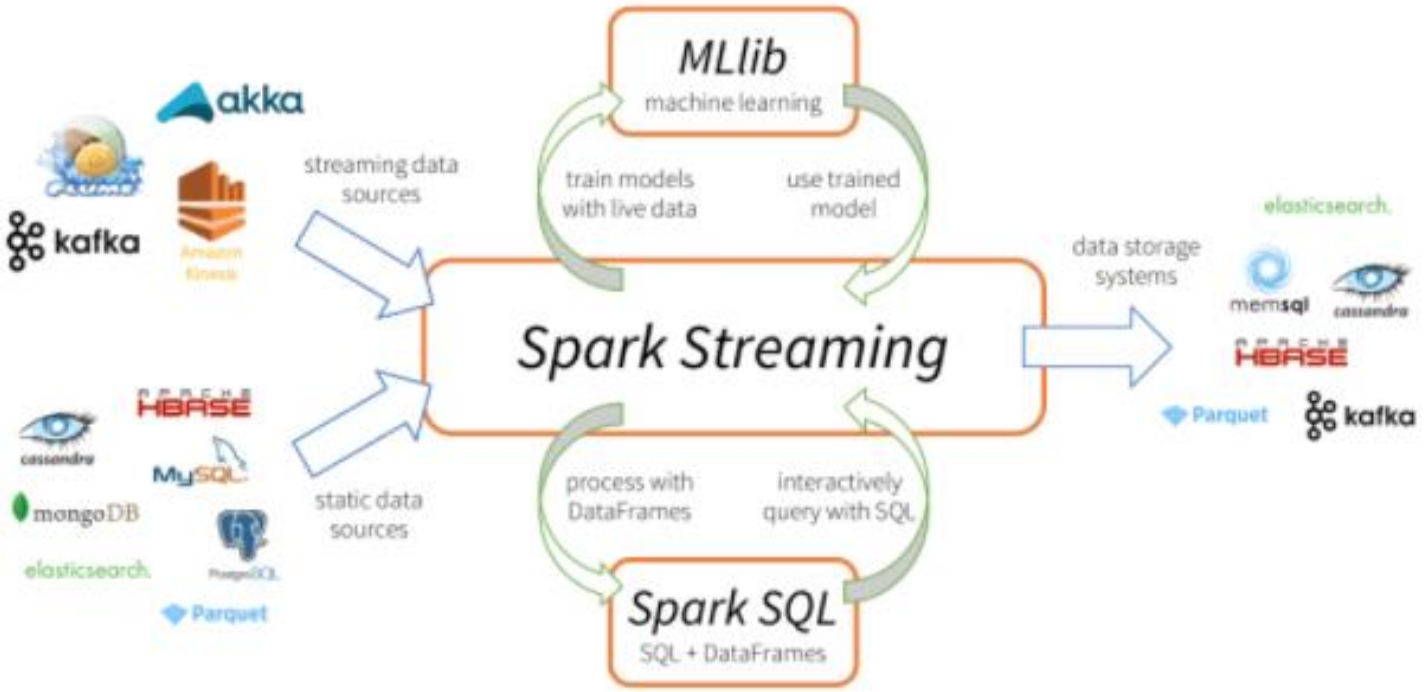
Apache Spark Streaming

- 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 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 >= 1.2)

Exemplu

Numărarea cuvintelor din date de tip text primite de la un server de date care ascultă la un socket TCP.

```
from pyspark import SparkContext
from pyspark.streaming import StreamingContext

sc = SparkContext("local[2]", "NetworkWordCount")
ssc = StreamingContext(sc, 1) # batch interval = 1 sec

# Crearea unui DStream care se va conecta la hostname:port
(localhost:9999)
lines = ssc.socketTextStream("localhost", 9999)

# Se imparte fiecare linie in cuvinte
words = lines.flatMap(lambda line: line.split(" "))
```

Exemplu (continuare)

```
# Numara fiecare cuvant din fiecare batch
pairs = words.map(lambda word: (word, 1))
wordCounts = pairs.reduceByKey(lambda x, y: x + y)

# Afiseaza primele 10 elemente din fiecare RDD generat in acest
Dstream la consola
wordCounts.pprint()

ssc.start()             # Porneste procesarea
ssc.awaitTermination()  # Asteapta terminarea procesarii
```

Exemplul necesita rularea Netcat:

```
$ nc -lk 9999
```

Si executarea programului Python precedent astfel:

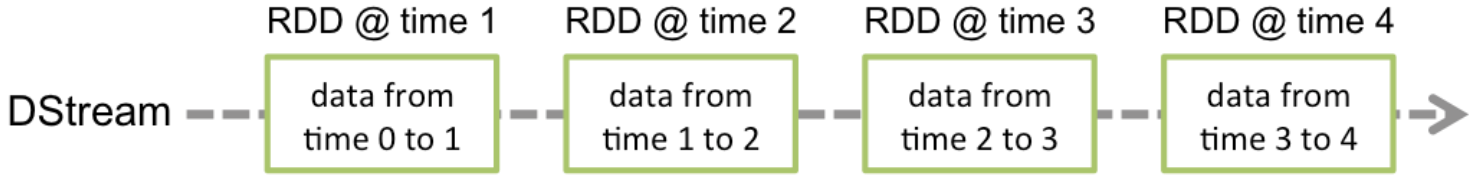
```
$ ./bin/spark-submit
examples/src/main/python/streaming/network_wordcount.py
localhost 9999
```

Concepte

- *StreamingContext* – necesar pentru a inițializa un program Spark Streaming, obiectul de acest tip fiind punctul de intrare al Spark Streaming
- Se poate crea pe baza *SparkContext*
- *DiscretizedStreams* (*DStreams*) – reprezintă un *stream* continuu de date, fie *stream*-ul de date de intrare primit de la sursă, fie cel de date procesate, obținute ca urmare a transformării *stream*-ului de intrare.

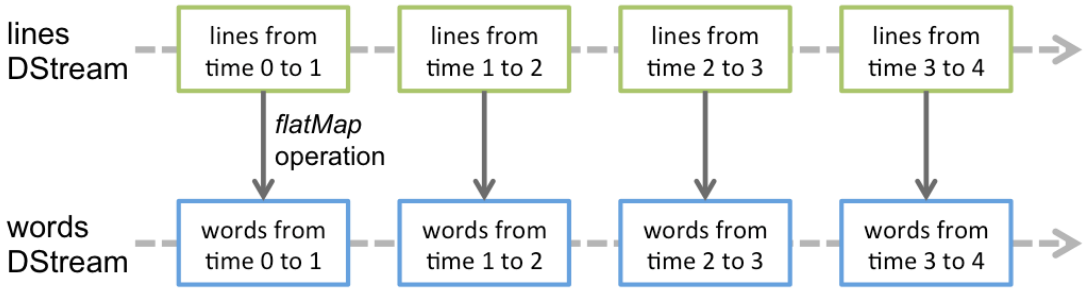
Concepte

- *DStream* – o serie continuă de RDD-uri
- Fiecare RDD din *DStream* conține date dintr-un anumit interval:



Concepte

- Orice operație aplicată pe un *DStream* se traduce în operații asupra RDD-urilor componente. În exemplul referitor la convertirea unui *stream* în cuvinte, operația *flatMap* este aplicată pe fiecare RDD din *DStream*-ul *lines*, generându-se RDD-urile *Dstream*-ului *words*.



- Transformările pe RDD-uri sunt efectuate de către *engine*-ul Spark. Operațiile *DStream* ascund aceste detalii, furnizând o API de nivel înalt.

Diferențe Structured Streaming vs. Spark Streaming

- *Spark Streaming* – librărie separată în Spark pentru procesarea datelor de tip *streaming*. Folosește DStream API, care se bazează pe RDD.
- Structured Streaming – construit peste librăria Spark SQL. Folosește DataFrame API. => Putem aplica cereri SQL sau operații din DataFrame API.
- Real Streaming? Structured Streaming > Spark Streaming
- RDD vs DataFrame? Structured Streaming (API-uri mai bune și mai optimizate) > Spark Streaming
- Tratarea event-time a datelor întârziate? Structured Streaming > Spark Streaming
- Garanții End-to-End? Structured Streaming – furnizeaza sintaxa end-to-end, exactly once
- Structured Streaming este mai apropiat *de real-time streaming*, iar Spark Streaming de procesarea de tip *batch*.

Bibliografie

1. Apache Spark documentation (https://spark.apache.org/docs/latest).

2. https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html

3. https://spark.apache.org/docs/latest/streaming-programming-guide.html

4. https://databricks.com/blog/2016/07/28/structured-streaming-in-apache-spark.html

5. https://dzone.com/articles/spark-streaming-vs-structured-streaming