Expert Services

Hands On Advanced Analytics with Apache Spark

Curs IV

Procesarea Distribuită

Not one for all, but all for one

Expert
Services

Data Clusters

It's usually someone else's problem

Expert
Services

Cum stocăm din ce în ce mai multe date?

1

Începem cu o nevoie simplă: spațiu pentru a stoca date. Soluția inițială? Un sistem mare cu multe hard disk-uri

2

Adăugăm și adăugăm tot mai multe date, simultan crescând și sistemul adăugând și adăugând tot mai multe hard disk-uri. Acest proces se numește Scalare Verticală.

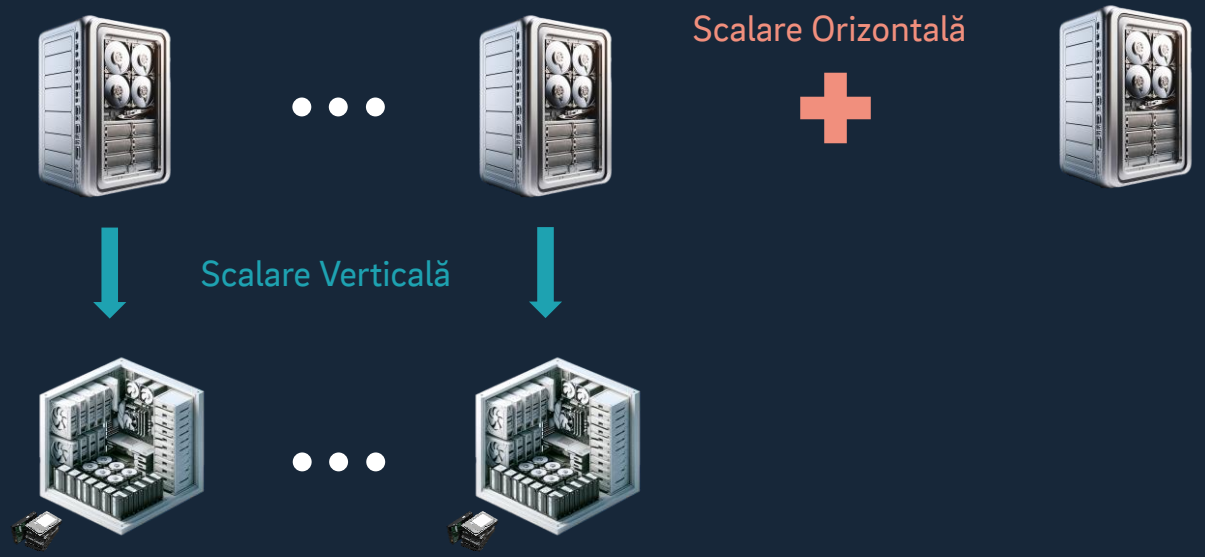
3

Adăugăm și adăugăm date, dar până la urmă, spațiul se epuizează. Nu mai încap hard disk-uri pe sistem. Ce facem? Dacă mai multe hard disk-uri nu mai încap, mai adăugăm un sistem. Acest proces se numește Scalare Orizontală.

Scalare Verticală <-> Orizontală

4

Cu cât continuăm să adăugăm mai multe și mai multe date, la un moment dat vom rămâne din nou fără loc pentru ele, și vom fi nevoiți să scalăm, să mărim capacitatea, fie vertical, dacă tehnologia ne permite, fie orizontal.



Cluster de Date

5

Întrucât ajungem la limita Scalării Verticale destul de rapid, sistemele de big data vor apela de cele mai multe ori la Scalare Orizontală. Astfel, majoritatea sistemelor de big data întotdeauna vor fi formate din mai multe mașini pentru stocarea de date.

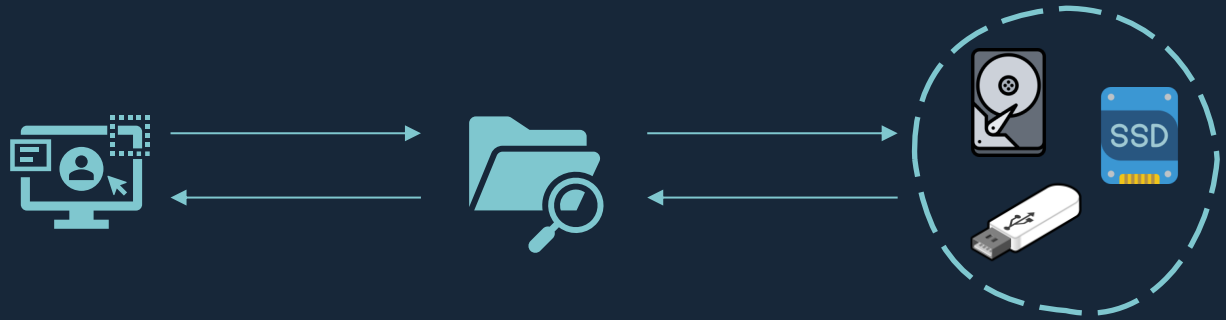
Acest ansamblu îl numim un cluster sau specific, cluster de date.

Un cluster poate fi format si doar dintr-o singură mașină, dar de regulă ne referim la un ansamblu când vorbim de clustere.

Interfață de stocare

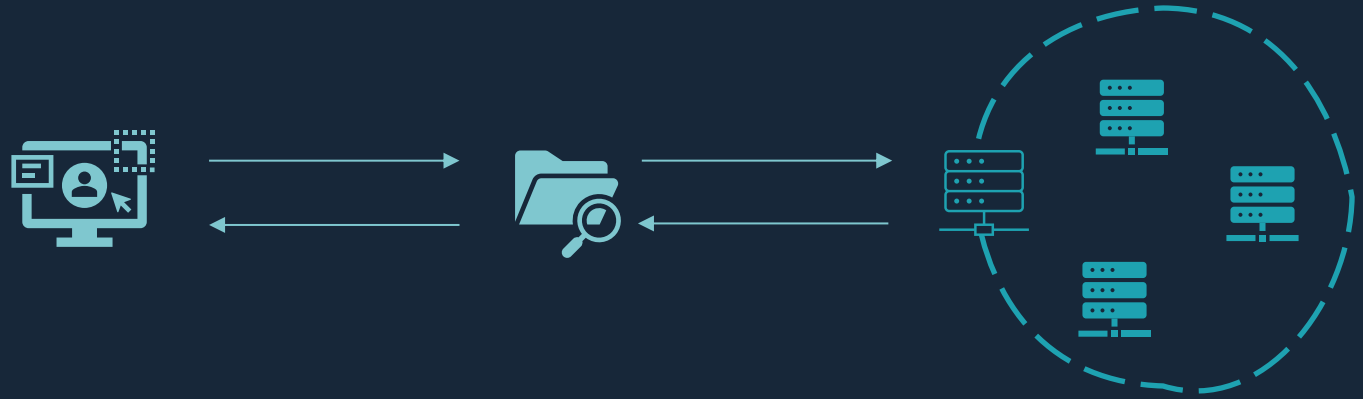
Mediile de stocare, cum ar fi hard disk-urile, SSD-urile și stickurile USB, sunt esențiale pentru operarea calculatoarelor, permițând păstrarea și consultarea datelor și fișierelor.

Pentru a uniformiza accesul și organizarea datelor, sistemele de operare utilizează un sistem de fișiere (filesystem), o interfață standardizată și eficientă de stocare, care facilitează comunicarea între programe și medii de stocare.



Interfață de stocare distribuită

Așa cum mediile de stocare sunt esențiale pentru operarea sistemului de operare și a programelor clasice, așa și clusterele sunt esențiale pentru operarea motoarelor Big Data, permițând păstrarea și consultarea datelor și fișierelor.

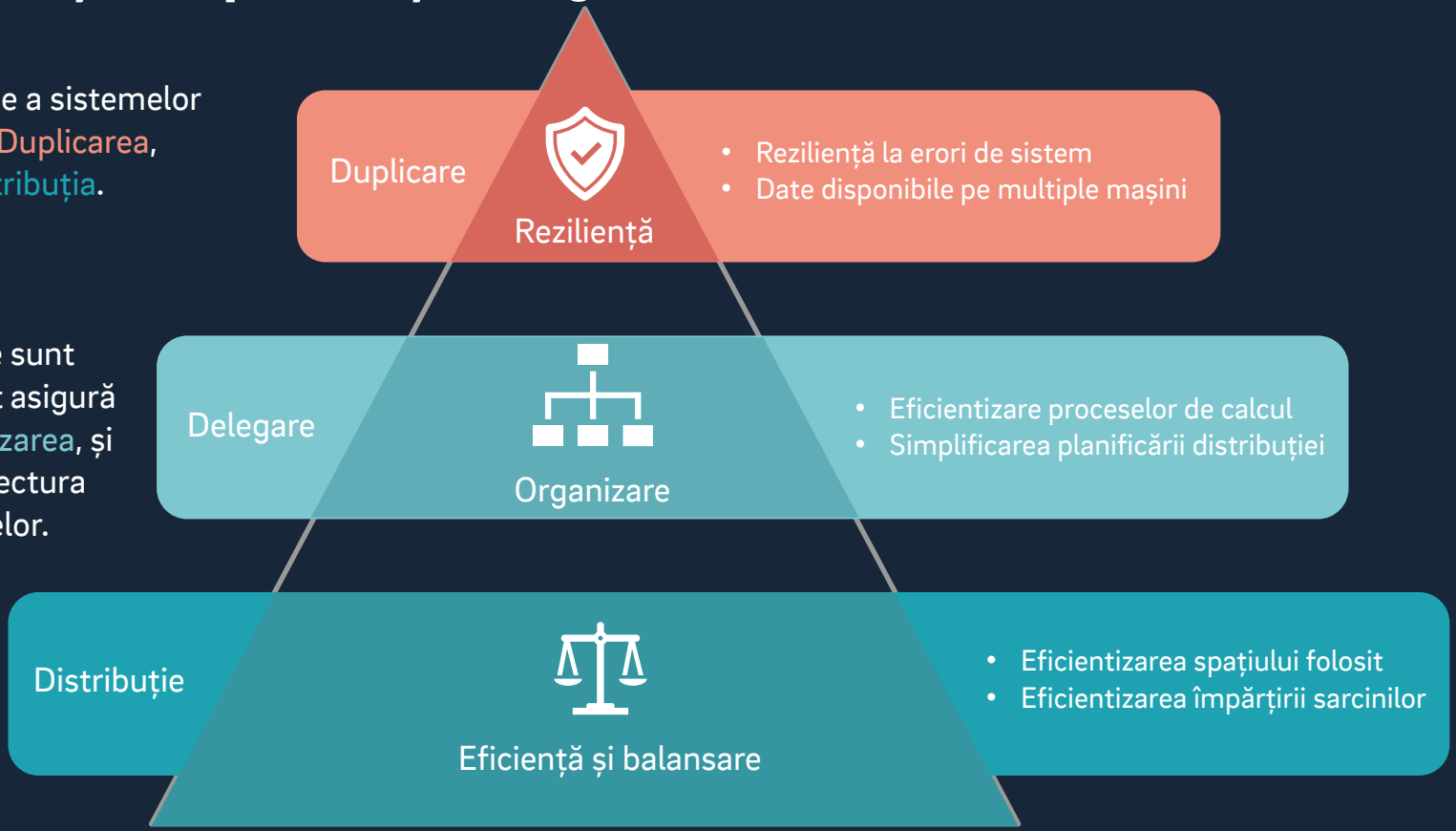


Precum sistemele de operare utilizează un sistem de fișiere pentru medii de stocare, așa și motoarele Big Data utilizează un sistem de fișiere distribuit (distributed filesystem) pentru clusterele de date, o interfață standardizată și eficientă pentru stocarea datelor, care facilitează comunicarea între motoare și clustere de date.

3D - Distribuție, duplicare și delegare

Trei principii cheie a sistemelor distribuite sunt: Duplicarea, Delegarea și Distribuția.

Aceste elemente sunt cruciale, întrucât asigură reziliența, organizarea, și eficiența în arhitectura distribuită a datelor.



Distribuția datelor prin sistem

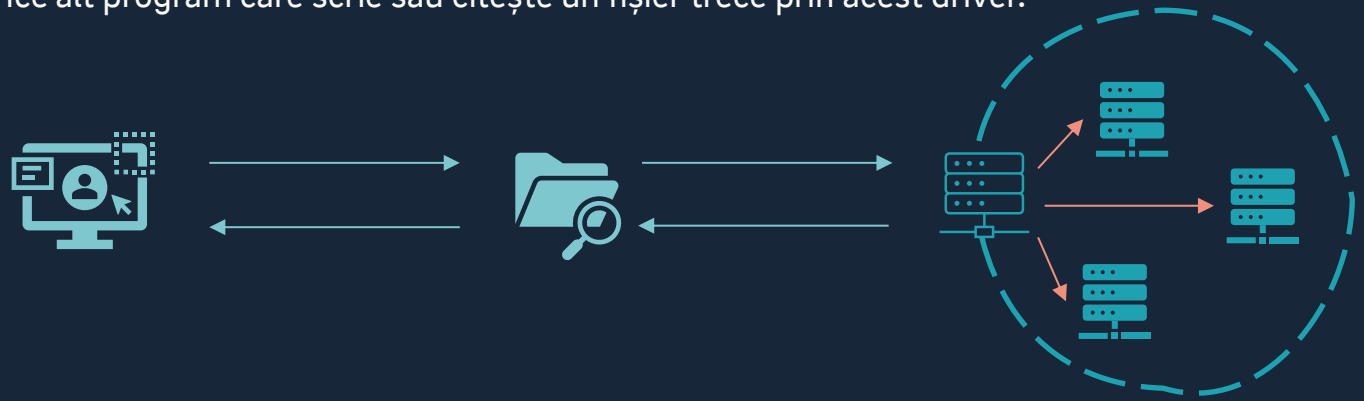
În clustere de date, partiționare, împărțirea fișierelor în bucăți, este întotdeauna recomandată pentru a asigura o distribuție egală a datelor pe întreg ansamblul de mașini, problemă pe care nu o are sistemul de fișiere tradițional care se ocupă doar de gestionarea unei singure mașini.

- Asta nu numai că permite ca fișierele să fie împărțite în bucăți care să se potrivească eficient în spațiile libere, dar nu trebuie transferat tot fișierul când avem nevoie de doar o parte din el.

Partiții mai mici înseamnă mai mult timp petrecut mergând prin mediul de stocare în loc de transferarea lor. În schimb, partiții mai mari înseamnă mai mult timp petrecut transferând date fără a fi nevoie și e posibil ca o partiție să fi încăput pe o mașină dacă partiția ar fi fost mai mică. Mărimea se alege în funcție de situație.

Delegare

Un sistem de fișiere tradițional operează prin intermediul unei singure căi de acces, programul de driver de pe mașină. Orice alt program care scrie sau citește un fișier trece prin acest driver.



Un sistem de fișiere distribuit poate oferi multiple puncte de acces care de obicei sunt și responsabile de organizarea fișierelor, unde se află ce parte din fișier. Acestea delegă mai departe comenzile spre celelalte mașini.

Aceste puncte de acces le numim Master sau Master Node.

Mai puține puncte de acces înseamnă o securitate mai sporită, mai puțini vector de atac. În schimb, mai multe puncte de acces oferă o mai bună distribuție a încărcării când sunt multe solicitări asupra clusterului.

Duplicarea datelor

Într-un sistem de fișiere obișnuit, prevenirea pierderii datelor în caz de defecțiune nu este o prioritate majoră, așadar datele nu sunt clonate decât la cerere, dar într-un cluster, unde nu vrem să pierdem date importante, același date dintr-un fișier sunt duplicate pe mai multe mașini din cluster.

În caz de o eroare la una dintre mașini, datele în continuare se află pe cluster.

Mai multe copii a datelor înseamnă o rezilienta la erori mai sporită, dar se ocupă mai mult spațiu de stocare. În schimb, mai puține copii a datelor reduce rezilienta la erori, dar se ocupă mai puțin spațiu de stocare suplimentar.

Dacă fișierele sunt stocate partiționat pe cluster, atunci fiecare partiție este clonată.

HDFS - Hadoop Distributed File System

HDFS este un sistem de stocare distribuit, folosit in special de programe Apache, precum Apache Hadoop și Apache Spark. Este unul dintre cele mai populare și versatile implementări open-source întrucât oferă toate caracteristicile principale.

Distribuție

Partiții de 64 / 128 / 256 MB

Delegare

O mașină numită Name Node și restul numite Data Node.

Duplicare

Fiecare partiție este duplicată pe 3 mașini diferite.

HDFS se folosește de sistemul de fișiere tradițional pentru a stoca datele pe fiecare mașină.

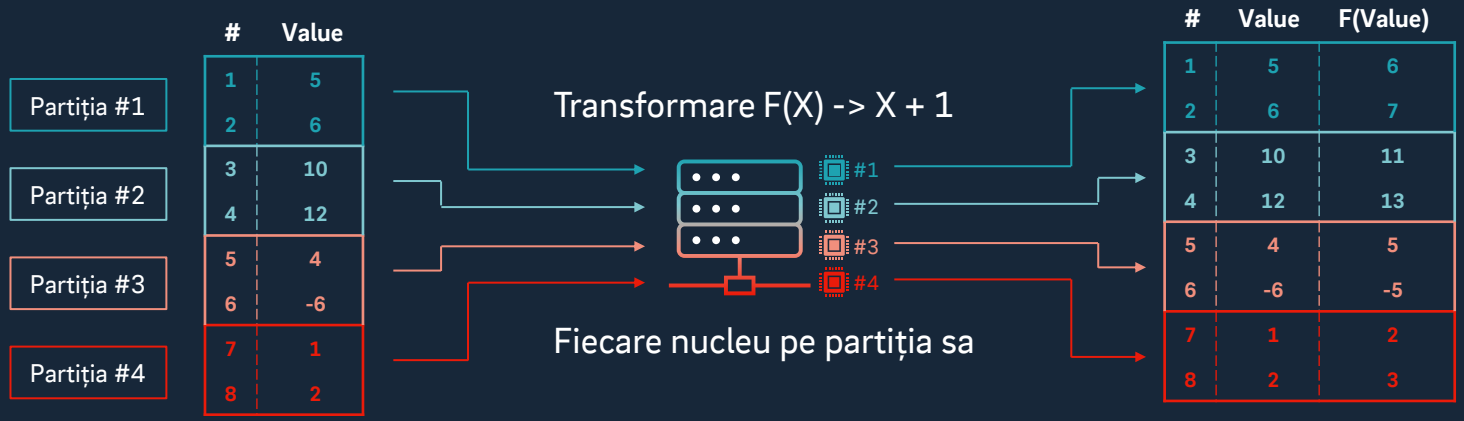
Procesare Paralelă

Twice the result with half the effort

Expert Services

Paralelism și Puterea Calculului Distribuit

Cel mai de bază exemplu, și originile procesării distribuite, sunt firele de execuție, utilizarea simultană a nucleelor unui procesor pentru a efectua calcule pe câte o partiție de date. De aici se începe povestea procesării distribuite.



În acest exemplu, pentru fiecare element dintr-un șir se calculează, cu cele 4 nuclee ale procesorului, în paralel:

o funcție de transformare a datelor - F(X)

În acest fel execuția este de 4 ori mai rapidă decât dacă am fi mers normal prin toate elementele și am fi calculat F.

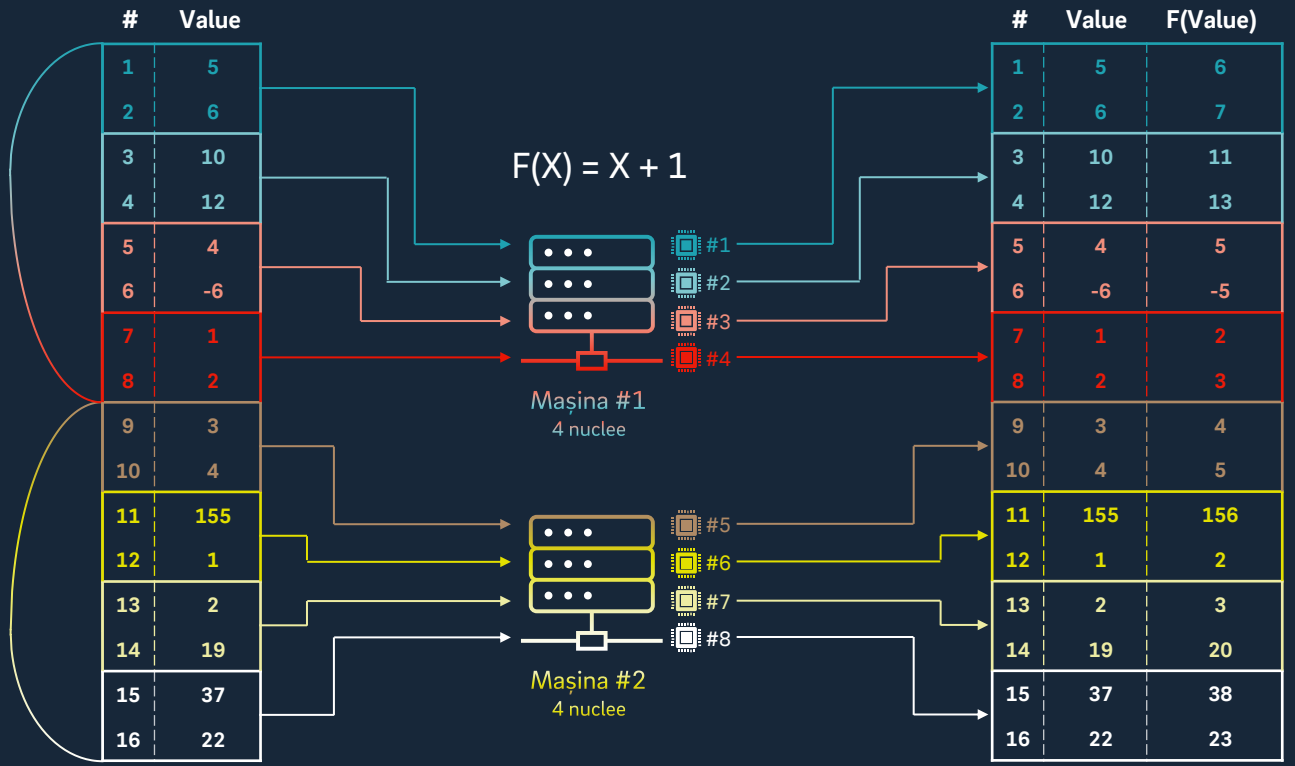
Dincolo de Calculul Individual

Trecem la nivelul următor, utilizarea mai multor mașini. Metoda rămâne aceeași, dar se adaugă transferul de date.

Este recomandat ca aceste **partiții de date** să se afle pe Mașina #1, dacă nu se va efectua un **transfer de date** de la Mașina #2 care va irosi timp.

În general, fiecare mașină îți procesează doar datele pe care pe stochează pentru a evita transferul de date care este "scump".

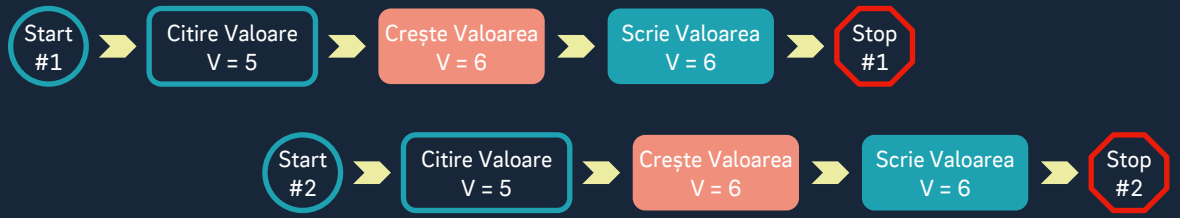
Este recomandat ca aceste **partiții de date** să se afle pe Mașina #2, dacă nu se va efectua un **transfer de date** de la Mașina #1 care va irosi timp.



Problema Sincronizării

Procesarea paralelă este dificilă. Nu este întotdeauna posibil să distribuim calculele în mod clar și ordonat. Principala problemă care apare este sincronizarea între firele de execuție, dependența unui fir de execuție de rezultatul altui fir de execuție.

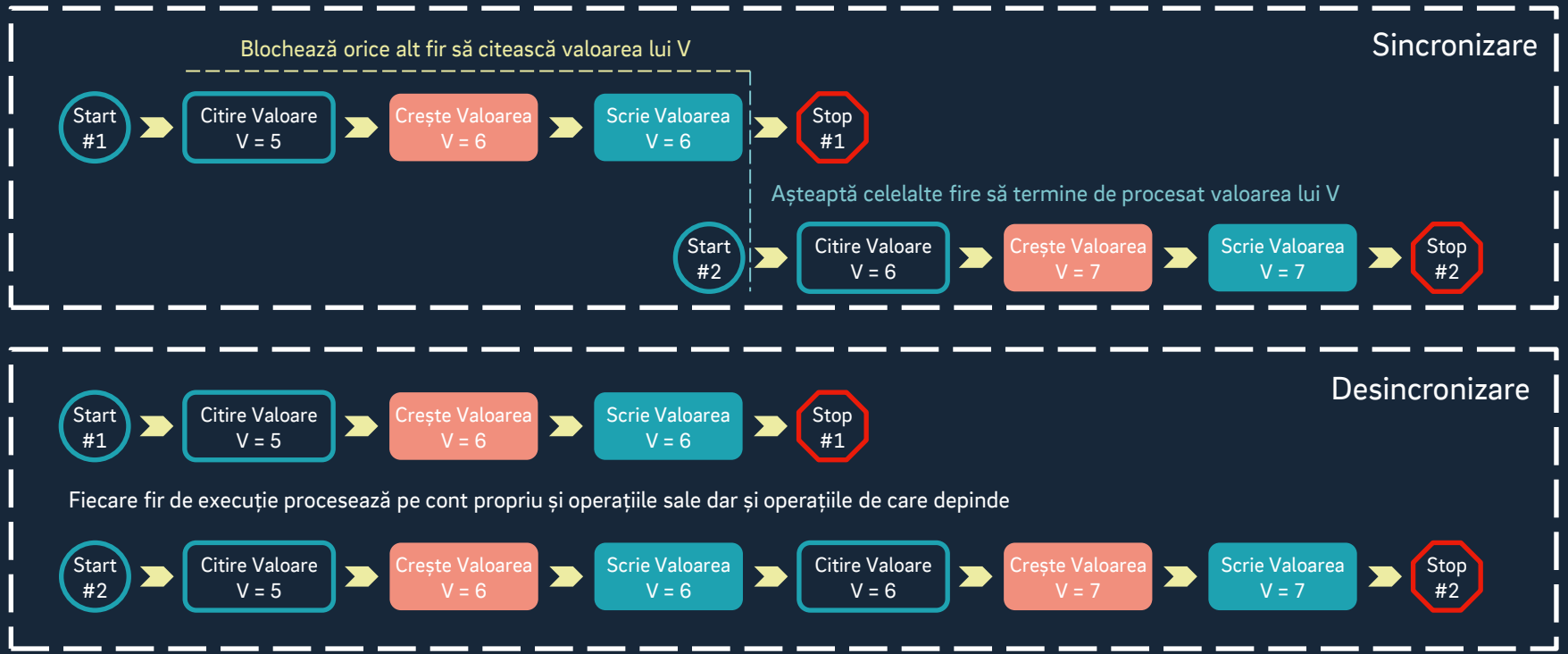
- Exemplu Problema Sincronizării - Executarea proceduri de incrementare a variabilei V în două fire de execuție



În acest exemplu, chiar dacă am apelat două de ori aceeași procedură de incrementare a valorii, la final aceasta ia valoarea 6 în loc de 7 = 5 + 2, deoarece firul de execuție #2 a început înainte ca firul de execuție #1 să scrie în V.

Sincronizare și desincronizare

Pentru a rezolva problema, avem două soluții: sincronizarea sau desincronizarea firelor de execuție.



Algoritmi distribuiți cu sincronizare / desincronizare

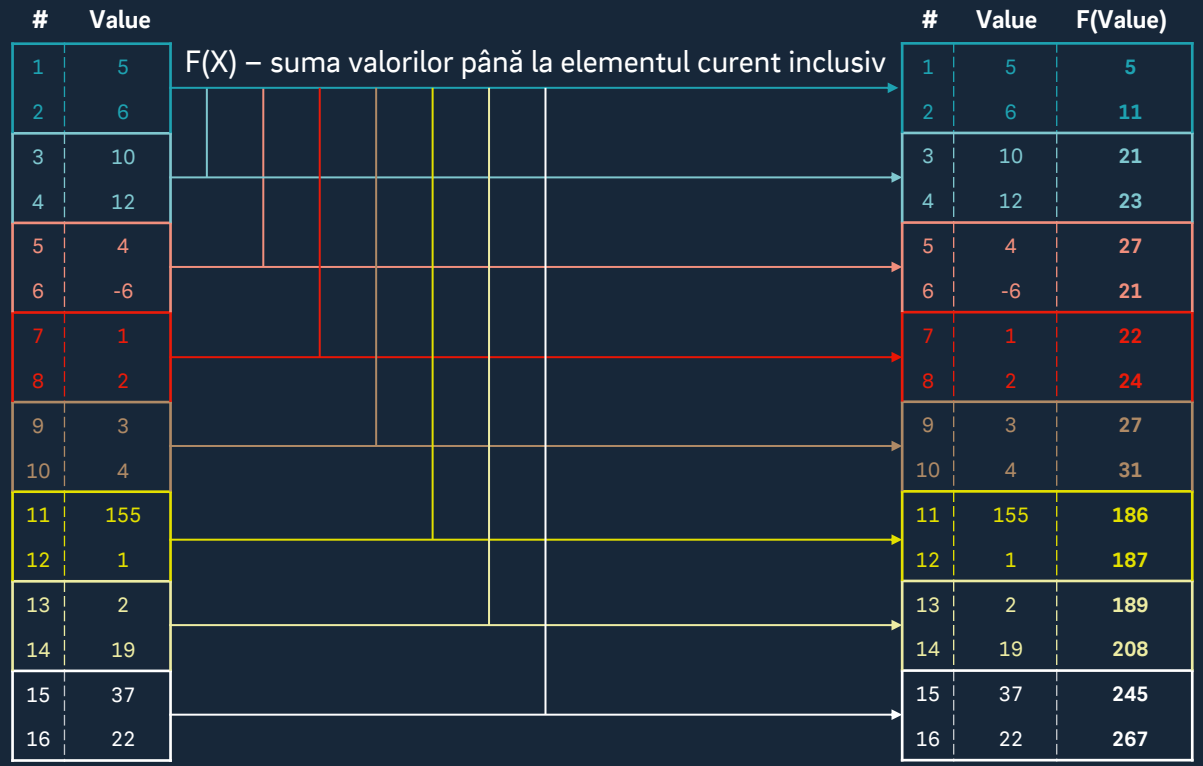
Un exemplu de algoritm în care apare problema sincronizării este cel de calculare a sumei cumulate.

- ✓ Tehnici sincronizare
- ❑ se așteaptă rezultatul anterior
  - ➢ timp pierdut în așteptare

- ✓ Tehnici desincronizate
- ❑ se calculează suma independent
  - ➢ calcul redundant

Cei mai optimizați algoritmi, în general, se folosesc de ambele tehnici, într-un mod eficient, tehnica potrivită la pasul potrivit.

Pentru această problemă, cel mai eficient algoritm are complexitatea:
O(n * log log n)

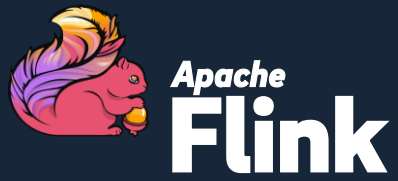


#	Value
1	5
2	6
3	10
4	12
5	4
6	-6
7	1
8	2
9	3
10	4
11	155
12	1
13	2
14	19
15	37
16	22

#	Value	F(Value)
1	5	5
2	6	11
3	10	21
4	12	23
5	4	27
6	-6	21
7	1	22
8	2	24
9	3	27
10	4	31
11	155	186
12	1	187
13	2	189
14	19	208
15	37	245
16	22	267

Motoare și Implementări Big Data

Motoarele Big Data sunt framework-uri care implementează aceste metode de lucru pentru a procesa seturile de date uriașe din Big Data. Scopul lor este de a face aceste metode accesibile pentru utilizare într-o gamă cât mai largă, astfel încât dezvoltatorii să poată realiza orice tip de operațiune necesară pentru prelucrarea seturilor de date.



Motoarele Big Data fie aduc propria implementare, fie folosesc o implementare cunoscută, dar toate oferă cel puțin o metodă de calcul desincronizat și una de calcul sincronizat, întrucât, în majoritatea cazurilor, ambele sunt necesare.

Generația 1 Map-Reduce: Scalarea Universală a Prelucrării Datelor

Una dintre aceste implementări este Map-Reduce, o metodă simplă de a scrie și efectua calcul în paralel, reducând complexitatea algoritmilor prin oferirea de doar două operații foarte eficiente: operația Map și operația Reduce.

Map – Calcul Desincronizat

#	Rows	
1	65	5
2	66	6
3	66	10
4	65	12
5	65	4
6	67	-6
7	69	1
8	68	2

#	Mapped Rows	
1	A	5
2	B	6
3	B	10
4	A	12
5	A	4
6	C	-6
7	E	1
8	D	2

Maparea (transformarea) fiecărui rând de date

Reduce – Calcul Sincronizat

#	Data	
1	A	5
2	B	6
3	B	10
4	A	12
5	A	4
6	C	-6
7	E	1
8	D	2

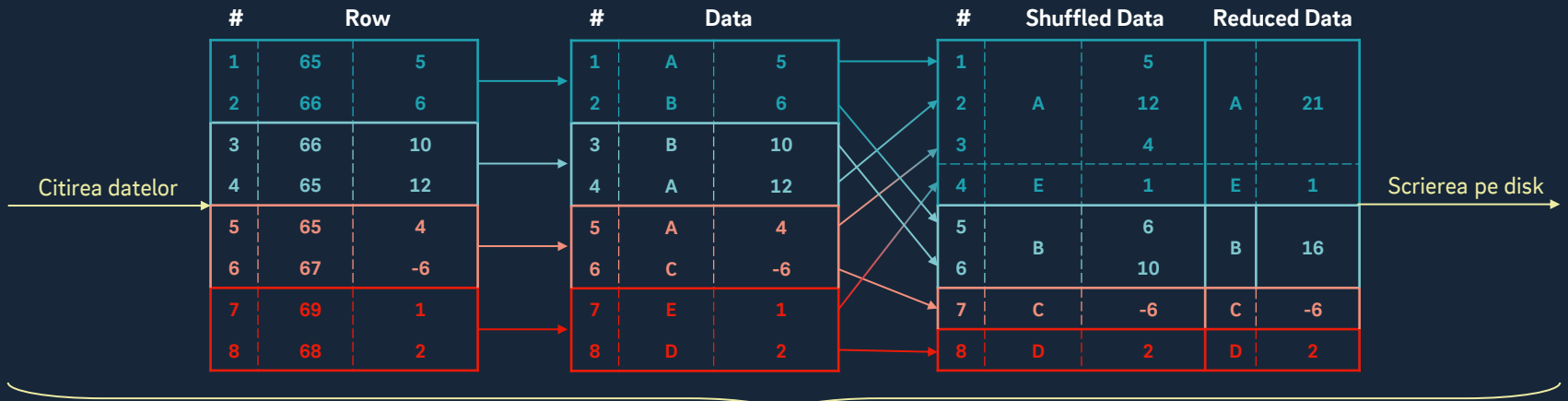
#	Shuffled Data		Mapped Data	
1	A	5	A	21
2		12		
3		4		
4	E	1	E	1
5	B	6	B	16
6		10		
7	C	-6	C	-6
8	D	2	D	22

Gruparea datelor după o cheie și reducere (agregarea) lor

Cu aceste două operații, putem scrie transformări eficient, în special cu operația Map, și agregări, cu operația Reduce.

Apache Hadoop: Fundamentul Prelucrării Big Data

Apache Hadoop este primul motor Big Data care a folosit paradigma Map-Reduce. Într-un program folosind Apache Hadoop, motorul citește datele de pe disk / cluster, execută o operație de map, și opțional una de reduce - ambele definite de dezvoltator - și apoi salvează rezultatele pe disc / cluster HDFS.



Un program Apache Hadoop.

Analizele de date detaliate se obțin prin executarea secvențială, una după alta, a mai multor astfel de programe.

Limitările Apache Hadoop

Implementarea Map-Reduce din Apache Hadoop este limitată. Analizele complexe necesită cod nou și deseori împărțirea în mai multe programe, cu scrieri costisitoare pe disc după fiecare pas.

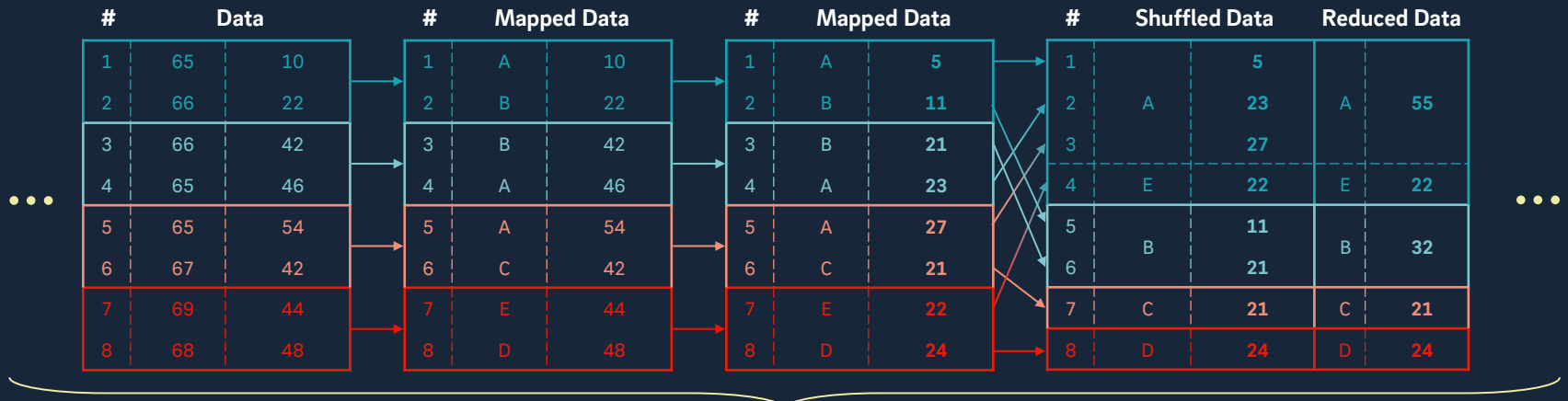
Avantaje	Dezavantaje
➢ Simplu și ușor de distribuit ❖ Map – fiecare fir de execuție procesează partiția sa de date ❖ Reduce – transfer de date paralel, fiecare fir de execuție procesează datele sale ➢ Rulează pe orice fel de hardware, fiind nativ Java ➢ Control asupra execuției fiecărui pas ➢ Integrare puternică cu HDFS	➢ Inflexibil ❖ Maxim o operație de Map și una de Reduce, pentru mai multe filtrări și agregări trebuie să scriem mai întâi datele pe disk ➢ Orice operație necesită mult cod ❖ operațiile comune pe date precum filtrarea datelor, asocieri (joins) și grupări complexe trebuie scrise de la zero de fiecare dată ➢ Optimizarea codului ține de dezvoltator

Deși limitat, Hadoop a introdus tehnica Map-Reduce în analiza de date, arătând că prelucrarea volumelor mari este posibilă. În continuare, cercetarea a avansat, iar motoarele Big Data ce au urmat au depășit limitările Hadoop.

Apache Spark: Revoluționarea Prelucrării Big Data

Metoda Map-Reduce este elegantă pentru ca e simplă și de înțeles și de implementat. Dar, limitările Apache Hadoop erau evidente. Așadar, a apărut motorul de big data Apache Spark, care a venit cu multe îmbunătățiri:

- Oricâte operații map / reduce
- Funcții predefinite, cum ar fi sortarea
- Stocarea datelor în memorie între operații
- Stocare temporară în memorie / pe disk



Un program Apache Spark.

În plus, Apache Spark include de asemenea, extensii pentru operații SQL, date în timp real și învățare automata.

Componente Apache Spark

Spark Core

Motorul de Bază Spark pentru execuții paralele largi și procesarea datelor distribuite. Toate celelalte componente se bazează pe aceasta.

- Managementul memoriei
- Recuperare după erori
- RDD - Resilient Distributed Datasets
- Planificare, distribuire și monitorizare
- Integrare cu sisteme de stocare
- Lucru cu date nestructurate

Spark SQL

Modulul de Spark care se ocupă de procesarea datelor semi-structurate și structurate. La bază, folosește Spark Core în mod transparent.

- Integrare cu diverse baze și formate de date
- Data Frames - bazate pe RDD în spate
- Analize Complexe

Spark Streaming

Extensie la Spark Core și Spark SQL pentru procesarea datelor în timp real.

- Integrare cu diverse sisteme de stocare a fluxurilor de date
- Procesarea rapidă a volumelor mari de date în timp real

Spark Mllib

Librărie low-level de învățare automată, simplă de utilizat si scalabilă. Folosește Spark Core la bază.

- Clasificarea Datelor
- Clusterizarea Datelor
- Deep Learning

GraphX

Motorul lui Spark pentru calcul de grafuri și stocarea lor.

- Gestionarea datelor de tip graf
- Navigare prin rețele complexe

Spark Core

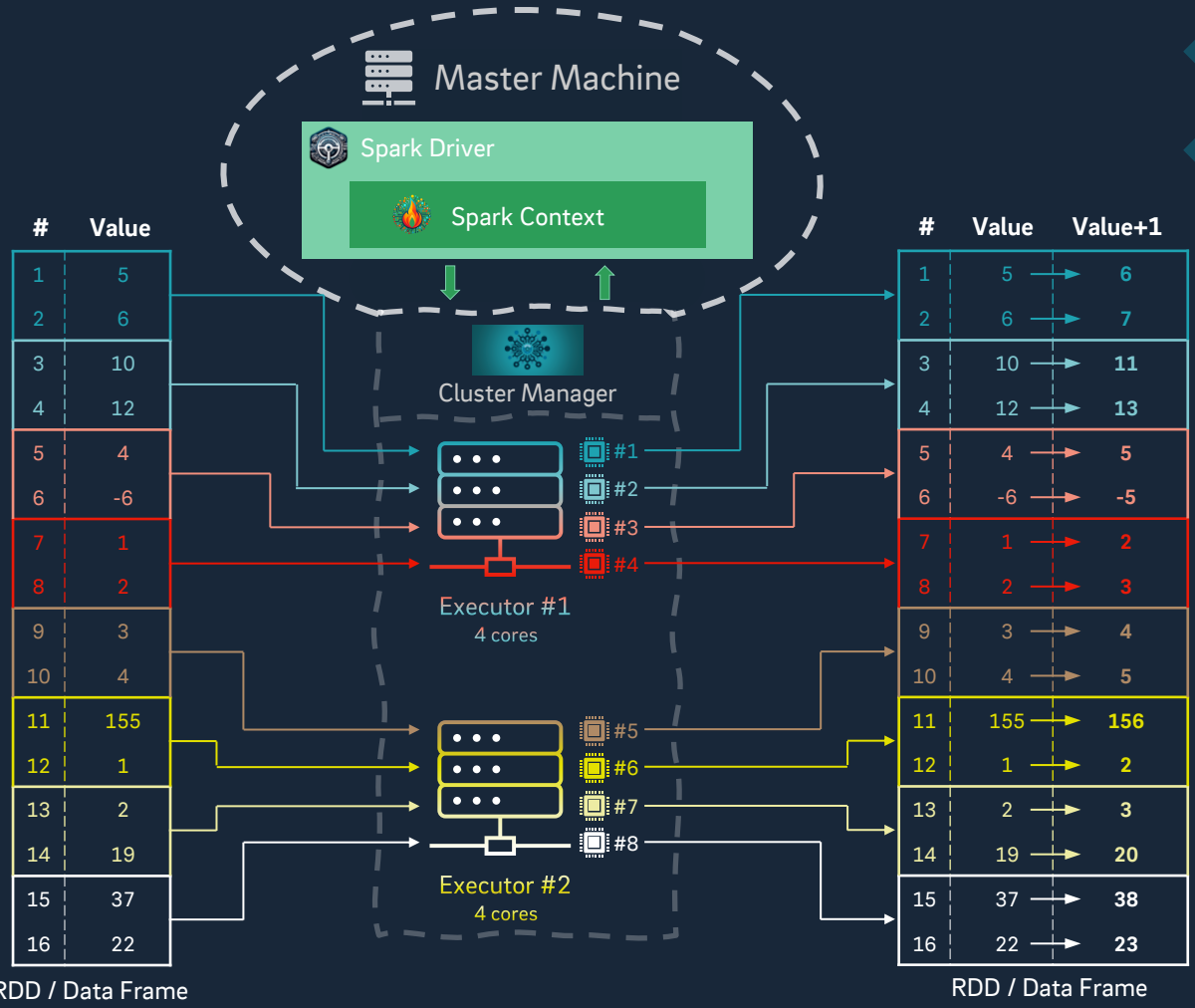
Not one for all, but all for one

Expert
Services

Arhitectura Spark

Spark folosește o arhitectură care se bazează pe o mașină principală, numită Master, care prin intermediul unui serviciu, numit Cluster Manager, dă ordine către mașinile care vor executa comenzile, numite Executor.

În procesul în care este rulat Spark, numit Spark Driver, dezvoltatorul va crea un obiect, fie el în Java, Python, etc., numit Spark Context, prin care librăria de Spark poate trimite instrucțiunile către executori.



Spark Core – PySpark

Spark Core este implementarea de bază din Spark. Extensiile precum Spark SQL folosesc această bază pentru a transmite și monitoriza operațiile de transformare a datelor pe cluster. Poate fi folosită și în mod direct de PySpark.

Spark Core PySpark oferă:

- Un obiectul de tip Spark Context, folosit pentru interacțiunea cu Spark.
- Un obiect de lucru cu date, numit RDD, oferind o interfață pentru executarea operațiilor Map-Reduce.
- Funcții gata implementate precum operația de filtrare / ordinare.
- Abilitatea de a construi propriile funcții de mapare / agregare în Python.
- Acces direct la partițiile interne ale setului de date.

Spre deosebire de Spark SQL, care suportă direct lucrul cu date semi-structurate sau structurate, precum JSON, CSV, Parquet, Spark Core este limitat la citirea de date în format text sau binar.

Spark Context

Spark Context este un obiect de Python oferit de librăria PySpark pentru Spark. Numai folosind acest obiect putem accesa tot ce oferă librăria. Orice extensie Spark folosește la bază acest obiect. Obiectul se creează astfel:

```
from pyspark import SparkConf
from pyspark.context import SparkContext

sc = SparkContext.getOrCreate(SparkConf().setMaster("local[*]"))
```

Se pot modifica și setările interne la crearea sesiunii dacă dorim:

```
from pyspark.sql import SparkSession, SparkConf

conf = SparkConf()
conf.set('spark.driver.memory', '3g’)

sc = SparkContext('local[*]', 'My App', conf=conf)
```

Dacă avem mai multe setări, adăugăm la lanț câte un apel la funcția config pentru fiecare setare. O listă cu toate opțiunile disponibile găsiți la https://spark.apache.org/docs/latest/configuration.html sau o căutare pe Google.

RDD - Resilient Distributed Dataset

Spark Core folosește un tip de obiect, numit RDD, pentru a gestiona un set de date. Acest obiect RDD reprezintă defapt o listă de obiecte, fie ele text, numere sau chiar alte liste (recomandat să fie cu același număr de valori).

- Crearea unui RDD Spark dintr-o listă de liste Python. În acest caz, toate rândurile vor fi liste cu același număr de valori.

```
data = [
   ['Vali', 23, 'Programator', 4, None, 'A', ['Sport', 'Boardgames']],
   ['Vlad', 34, 'Instalator', 11, None, 'B',['Alergare']],
   ['Bea', 29, 'Reporter', 7, True, 'B', None]
]

data_df = sc.parallelize(data)
```

Fiind concepute pentru Big Data, un RDD **nu** încarcă datele. El reține doar locația datelor. La fiecare operație de transformare (map / reduce) de date, un nou RDD va fi creat care reține locația și tipul rândului, precum și lanțul de transformări. Transformările vor fi efectuate doar în momentul scrierii sau colectării datelor de către executori.

Citirea Datelor

În Big Data, în general, datele sunt stocate extern în unul sau mai multe formate. PySpark Spark Core oferă metode pentru citirea datelor în format binar / text pe care dezvoltatorul le poate ulterior parsa.

- Citirea unui fișier text. O linie a fișierului / rând din RDD.

```
data_rdd = sc.textFile('/path/to/text/file')
```

- Citirea fișierelor unui folder. Un întreg fișier / rând din RDD.

```
data_rdd = sc.wholeTextFiles('/path/to/text/file')
```

De asemenea, fișierul sau fișierele unui folder pot fi citite și în format binar direct.

- Citirea unui fișier / folder direct în format binar. Un întreg fișier / rând din RDD.

```
data_rdd = sc.binaryFiles('/path/to/file/or/folder')
```

Spark Context are multe funcții pentru diverse operațiil. O listă cu toate opțiunile disponibile găsiți la https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.SparkContext.html#pyspark.SparkContext.

Data Frame – Colectarea Datelor

De multe ori ne găsim în situația în care este nevoie să depanăm procesul de transformare a datelor. Așadar, Spark ofertă metode de colectare a datelor pentru acest scop, dar și pentru prelucrarea ulterioară cu alte librării.

- Pentru a colecta datele într-o listă de Python, folosim:

```
data_list = data_rdd.collect()
```

- Pentru a colecta datele într-un dicționar atunci când rândurile au doar două elemente:

```
data_pandas_pdf = data_df.collectAsMap()
```

Atenție! Datele sunt transferate mai întâi de la executori pe Spark Driver la colectare. Dacă sunt prea multe date, există pericolul ca procesul de Spark Driver să nu mai facă față și să fie terminat de către sistem.

Scrierea Datelor

În Big Data, după procesare, datele sunt stocate. PySpark Spark Core oferă metode pentru scrierea în diverse formate de stocare de nivel jos, precum text.

- Scrierea fișierelor în format text.

```
data_rdd.saveAsTextFile('/path/to/save/folder/')
```

Fiecare partiție de date va fi scrisă în câte un fișier. Fiecare rând din partiție va fi scris pe câte o linie din fișier. Valorile rândului vor fi scrise despărțite prin spații.

Se pot scrie și formate mai sofisticate, formatul Pickle, foarte utilizat în cadrul Python:

- Scrierea fișierelor în format Pickle.

```
data_rdd.saveAsPickleFile('/path/to/save/folder')
```

RDD-urile au mai multe funcții pentru scrierea datelor. O listă cu toate opțiunile disponibile găsiți la https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.RDD.html.

Maparea Datelor - Map

Principalele metode de transformare a RDD-urilor sunt metodele din paradigma Map-Reduce precum operația map.

- Operația de Mapare / rând – Aplicarea unui funcții asupra fiecărui rând

```
new_data_df = data_rdd.map(lambda x: (x[0], x[1] + 1, x[6]))
```

- Un nou obiect de tip RDD este returnat care transformă fiecare rând după funcția furnizată. RDD-ul de la care a pornit maparea va rămâne neschimbat.

```
print(data_rdd.collect())
```

```
[
 ['Vali', 23, 'Programator', 4, None, 'A', ['Sport', 'Boardgames']],
 ['Vlad', 34, 'Instalator', 11, None, 'B', ['Alergare']],
 ['Bea', 29, 'Reporter', 7, True, 'B', None]
]
```

```
print(new_data_rdd.collect())
```

```
[
 ('Vali', 24, ['Sport', 'Boardgames']),
 ('Vlad', 35, ['Alergare']),
 ('Bea', 30, None)
]
```

Maparea Datelor – Flat Map

Spark Core oferă și o metodă de mapare care permite funcției furnizate să returneze mai multe rânduri.

- Operația de Mapare / un rând la mai multe rânduri – Aplicarea unui funcții asupra fiecărui rând

```
new_data_rdd = data_rdd.flatMap(lambda x: [ (x[0], x[1] + 1, consumer) for consumer in (x[6] or []) ])
```

- Un nou obiect de tip RDD este returnat format din rândurile returnate de funcția furnizată. RDD-ul de la care a pornit maparea va rămâne neschimbat. În acest caz, fiecare rând va returna câte două rânduri.

```
print(data_rdd.collect())
```

```
[
 ['Vali', 23, 'Programator', 4, None, 'A', ['Sport', 'Boardgames']],
 ['Vlad', 34, 'Instalator', 11, None, 'B', ['Alergare']],
 ['Bea', 29, 'Reporter', 7, True, 'B', None]
]
```

```
print(new_data_rdd.collect())
```

```
[
 ('Vali', 24, 'Sport'),
 ('Vali', 24, 'Boardgames'),
 ('Vlad', 35, 'Alergare')
]
```

Pair RDD - Resilient Distributed Dataset Key-Value

Pentru a putea efectua operațiile de shuffle / reduce, trebuie specificat pentru fiecare rând o câte o cheie de grupare. Așadar, din design-ul Spark, metodele de shuffle și reduce pot fi executate doar pe RDD-uri ale căror rânduri sunt liste / tuple formate din două valori, prima valoare fiind considerată cheia, iar a doua datele, o valoare / listă de valori.

- Folositea operației Map pentru a converti fiecare rând la o listă / tupla de două elemente

```
pair_rdd = data_rdd.map(lambda x: (x[5], x))
```

- Un nou obiect de tip RDD este returnat format din rânduri a câte două valori, prima fiind valoarea coloanei 5 din fiecare rând și a doua fiind datele originale. RDD-ul de la care a pornit maparea va rămâne neschimbat.

```
print(pair_rdd.collect())
```

```
[
 ('A', ['Vali', 23, 'Programator', 4, False, 'A', ['Sport', 'Boardgames']]),
 ('B', ['Vlad', 34, 'Instalator', 11, None, 'B', ['Alergare']]),
 ('B', ['Bea', 29, 'Reporter', 7, True, 'B', None])
]
```

Reducerea Datelor – Reduce - Shuffle & Agregări

Operația completă de reduce compresează valorile cu aceeași cheie la un singur rând. Această funcție repartiționează setul de date, grupând datele ce necesită reduse împreuna în aceeași partiție, după aplică funcția de reduce.

- Operația de Reduce – Agregarea unui grup într-un singur rând

```
new_data_rdd = pair_rdd.foldByKey(0, lambda aggregation_now, next_row: aggregation_now  + next_row[1])
```

- Este returnat un nou Pair RDD, unde prima valoare din fiecare rând este cheia, iar a doua valoare este rezultatul apelării funcției mai întâi pe valoarea inițială, în acest caz zero, și un rând din grup, apoi pe rezultat și al doilea rând din grup, ș.a.m.d. Metoda foldByKey poate fi apelată doar pe Pair RDD-uri.

```
print(pair_rdd.collect())
```

```
[
 ('A', ['Vali', 23, 'Programator', 4, False, 'A', ['Sport', 'Boardgames']]),
 ('B', ['Vlad', 34, 'Instalator', 11, None, 'B', ['Alergare']]),
 ('B', ['Bea', 29, 'Reporter', 7, True, 'B', None])
]
```

```
print(new_data_rdd.collect())
```

```
[
  ('A', 23),
  ('B', 63)
]
```

- Spark mai întâi combină toate datele cu aceeași cheie care se află deja în aceleași partiții, apoi repartiționează datele după cheie și combină rezultatele. În acest mod, se reduce numărul de date transferate la repartiționare.

Reducerea Datelor – Shuffle

Spark Core nu constrânge dezvoltatorul doar la operații de Map-Reduce. Este oferită și o metodă care doar repartiționează setul de date, grupând datele ce necesită reduse împreuna în aceeași partiție.

- Operația de Reconfigurare a Partițiilor – Repartiționarea datelor după valoarea cheie

```
repartitioned_pair_rdd = pair_rdd.partitionBy(2)
```

- Un nou obiect de tip RDD este returnat, cu numărul de partiții specificat. Rândurile cu aceeași valoare cheie vor fi mutate în aceeași partiție. Metoda poate fi apelată doar pe Pair RDD-uri întrucât se bazează pe cheie.

- Operația de Reconfigurare a Partițiilor – Repartiționarea datelor în mod aleatoriu

```
new_data_df = data_rdd.repartition(2)
```

```
new_data_df = data_rdd.coalesce(1)
```

- Un nou obiect de tip Data Frame este returnat cu numărul de partiții specificat, rândurile fiind distribuite în mod aliator, dar echilibrat, la partiții. Metoda poate fi apelată pe orice fel de RDD. Metoda coalesce combină partițiile existente, așadar poate fi apelată doar dacă se dorește micșorarea numărului de partiții a datelor.

Maparea și Reducerea Datelor – Map Partition

Cea mai flexibilă operație oferită de Spark Core este operația de mapare a partițiilor. Folosind această metodă, se poate transforma câte o partiție odată, sau în combinație cu partiționarea, se poate folosi pentru a agrega datele.

- Operația de Mapare a unei partiții – Comprimarea partiției într-un singur rând

```
new_data_rdd = data_rdd.mapPartitions(lambda rows: [(x[0], x[1], x[5]) for x in rows])
```

- Un nou obiect de tip Data Frame este returnat cu rândurile rezultate prin aplicarea funcției per fiecare partiție. Majoritatea metodelor de map (map / flatMap / reduce) se foloseste de această metodă.

data_df.show()

```
[
 ['Vali', 23, 'Programator', 4, None, 'A', ['Sport', 'Boardgames']],
 ['Vlad', 34, 'Instalator', 11, None, 'B', ['Alergare']],
 ['Bea', 29, 'Reporter', 7, True, 'B', None]
]
```

new_data_df.show()

```
[
 ('Vali', 23, 'A'),
 ('Vlad', 34, 'B'),
 ('Bea', 29, 'B')
]
```

Un program simplu PySpark Spark Core

- Pas 1: Create porții de acces către Spark SQL, Sesiune Spark

```
from pyspark import SparkContext; import json
sc = SparkContext.getOrCreate(SparkConf().setMaster("local[*]"))
```

- Pas 2: Citirea fișierelor de într-un Data Frame

```
data_rdd = sc.wholeTextFiles('/path/to/course/data/json/folder')
```

- Pas 3: Extragerea zonei și a consumatorilor

```
pair_rdd = data_rdd.flatMap(lambda x: [(json.loads(r)['zona'], json.loads(r).get('extra')) for r in x[1].splitlines()])
```

- Pas 4: Calcularea numărului de consumatori declarați per zonă

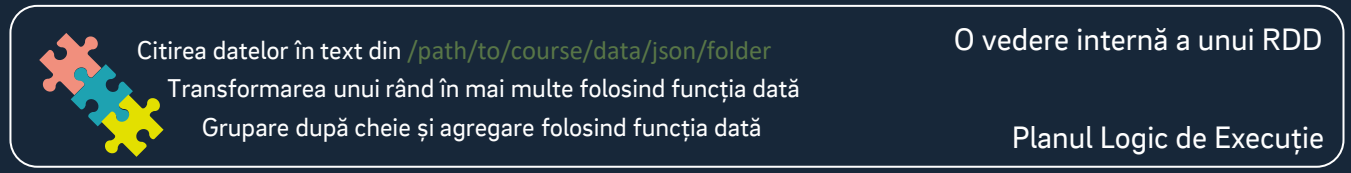
```
final_data_rdd  = pair_rdd.foldByKey(0, lambda aggregation, next_value: aggregation + len(next_value or []))
```

- Pas 5: Scrierea datelor finale într-un folder

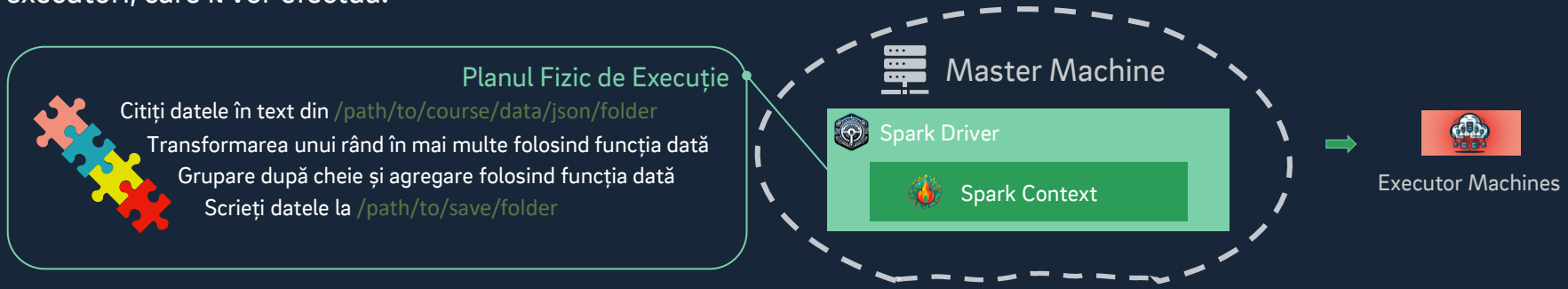
```
final_data_rdd.saveAsTextFile('/path/to/save/folder')
```

RDD – View și Lineage

Un RDD este defapt un „View", similar cu cel din bazele de date SQL, asupra datelor. Într-un astfel de obiect, se reține doar de unde provin datele și operațiile care trebuie efectuate, numite Planul Logic de Execuție.

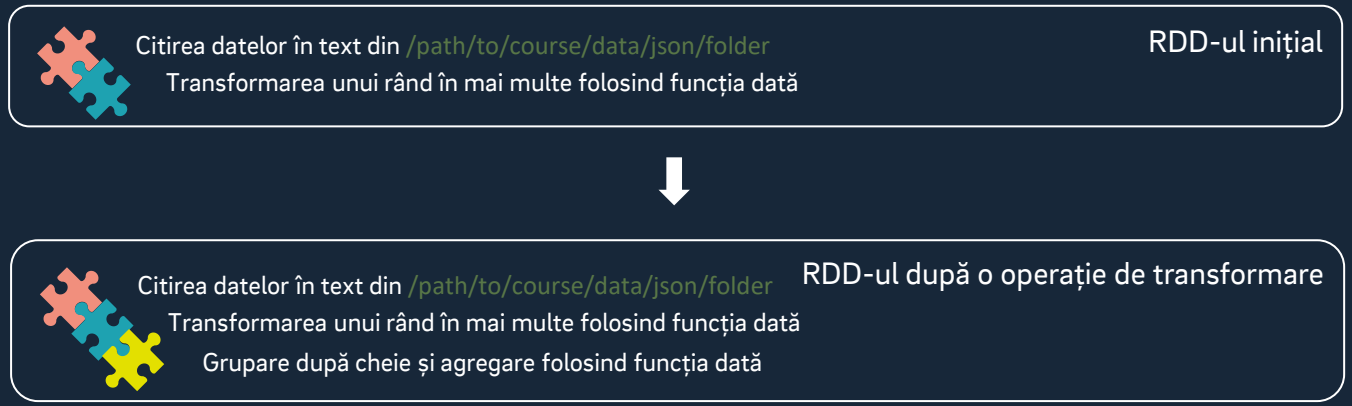


Calculele vor fi efectuate doar în momentul accesării acestui „View", adică atunci când dorim să le scriem, afișăm sau colectăm. În acel moment, mașina Master va trimite „comenzile", Planul de Execuție Fizic al RDD-ului, la executori, care îl vor efectua.



RDD – Imutabilitate

Întrucât un „View", RDD, nu poate fi modificat, toate funcțiile de transformare din PySpark întotdeauna vor returna un „View" nou, adică un nou RDD, clonă a celui anterior dar la care se adaugă noua transformare.

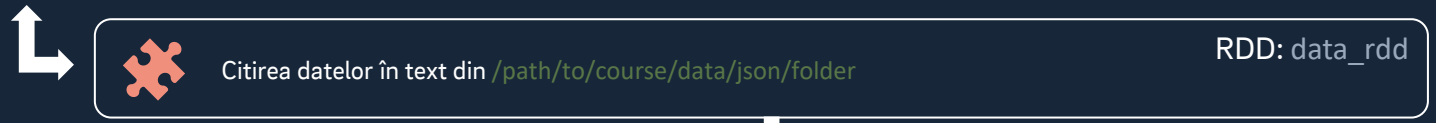


Defapt, nici o funcție din Spark nu permite modificarea unui obiect de tip RDD. Odată creat, un obiect de RDD nu mai poate fi modificat!

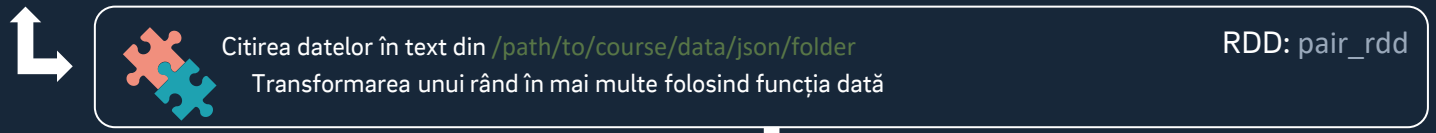
Această proprietate se numește în programare proprietatea de imutabilitate.

Un program simplu PySpark Spark Core – Lineage

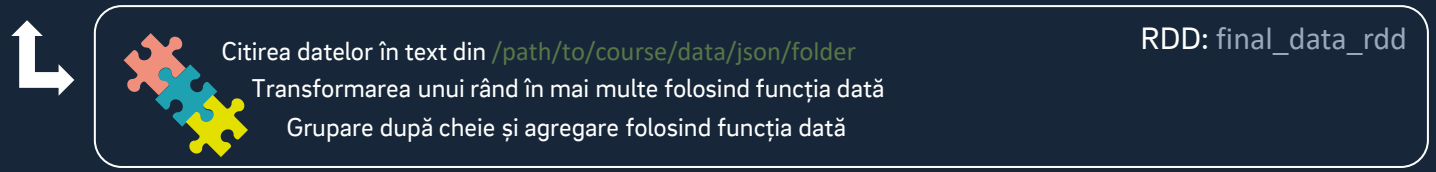
```
data_df = spark.read.format('json').load('/path/to/course/data/folder')
```




```
pair_rdd = data_rdd.flatMap(lambda x: [(json.loads(r)['zona'], json.loads(r).get('extra')) for r in x[1].splitlines()])
```




```
final_data_rdd  = pair_rdd.foldByKey(lambda aggregation, next_value: aggregation + len(next_value or []))
```



RDD – Afișarea Lineage-ului

De multe ori ne găsim în situația în care este nevoie să depanăm operațiile de transformare a datelor. Așadar, Spark Core ofertă metode pentru afișarea comenzilor și operațiilor care trebuie efectuate pe date.

- Pentru a printa planul logic de execuție la consolă, folosim:

```
print(final_data_rdd.toDebugString().decode())
```

```
(1) PythonRDD[193] at RDD at PythonRDD.scala:53 []
 |  MapPartitionsRDD[192] at mapPartitions at PythonRDD.scala:160 []
 |  ShuffledRDD[191] at partitionBy at NativeMethodAccessorImpl.java:0 []
 +-(1) PairwiseRDD[190] at foldByKey at [REDACTED].py:1 []
    |  PythonRDD[189] at foldByKey at [REDACTED].py:1 []
    |  /path/to/course/data/folder MapPartitionsRDD[184] at wholeTextFiles at NativeMethodAccessorImpl.java:0 []
    |  WholeTextFileRDD[183] at wholeTextFiles at NativeMethodAccessorImpl.java:0 []
```

Mai multe detalii găsiți la https://books.japila.pl/pyspark-internals/PythonRDD/ .

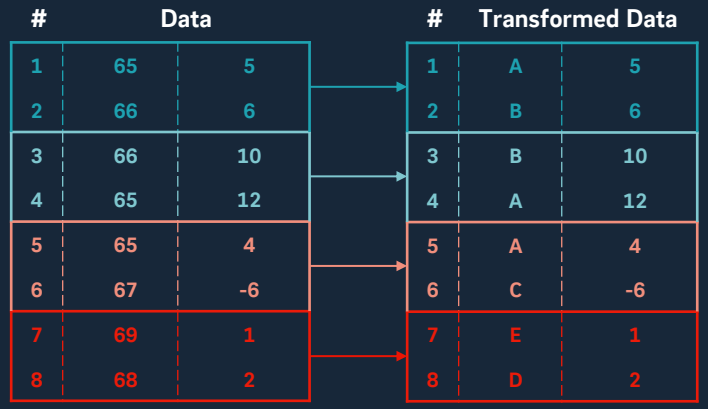
From Core to SQL and Back

Ce nu trebuie nu facem

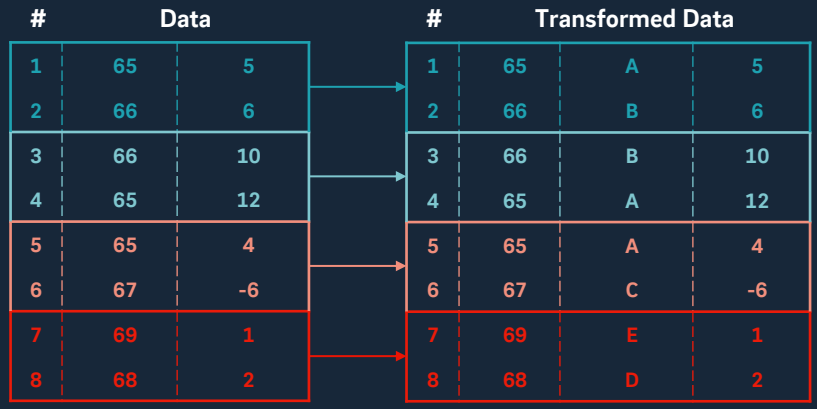
Expert Services

Data Frame -> Map-Reduce – Operațiile de transformare

Toate operațiile de transformare din ce pot fi aplicate peste un Data Frame din Spark SQL sunt în spate efectuate folosind operația de map din Map-Reduce.



Transformare a unei coloane existente



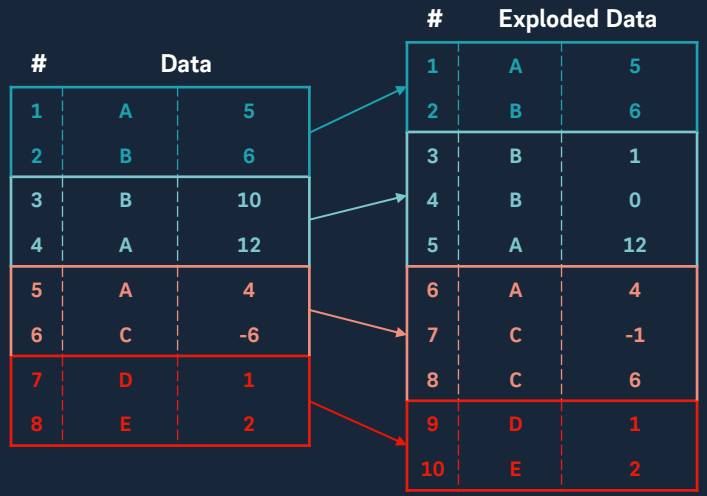
Adăugarea unei coloane noi

Când Spark SQL utilizează expresii și nu funcții definite de dezvoltator, codul de mapare nu numai că este generat direct în limbajul nativ Spark, dar și transformările consecutive sunt combinate în aceeași operație, sporind eficiența.

Data Frame -> Map-Reduce – Operațiile de filtrare / expandare

Operațiile de filtrare și cele de explozie ce pot fi aplicate peste un Data Frame din Spark SQL sunt și ele în spate efectuate folosind operația de map din Map-Reduce.

Filtrare datelor

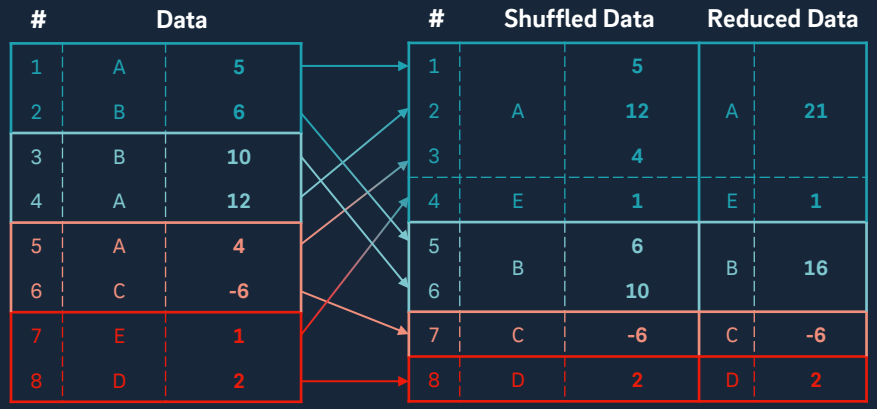


Expandare datelor

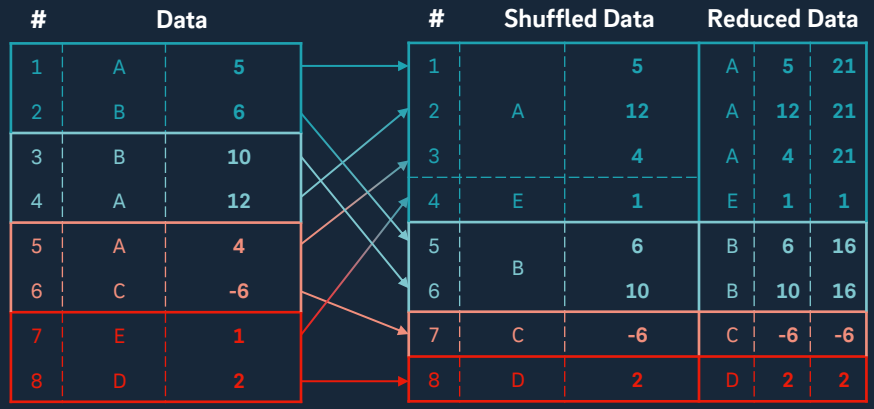
Precum transformările de date, codul de mapare a filtrărilor și expandărilor nu numai că este generat direct în limbajul nativ Spark, dar operațiile sunt și combinate cu alte transformări în aceeași funcție, sporind eficiența.

Data Frame -> Map-Reduce – Operațiile de grup

Operațiile de grupare, operația premergătoare agregărilor și funcțiilor window din Spark SQL, sunt în spate efectuate folosind operația de reduce din Map-Reduce, iar operațiile de agregare sau window folosind operația de map.



Operația de grupare și agregare

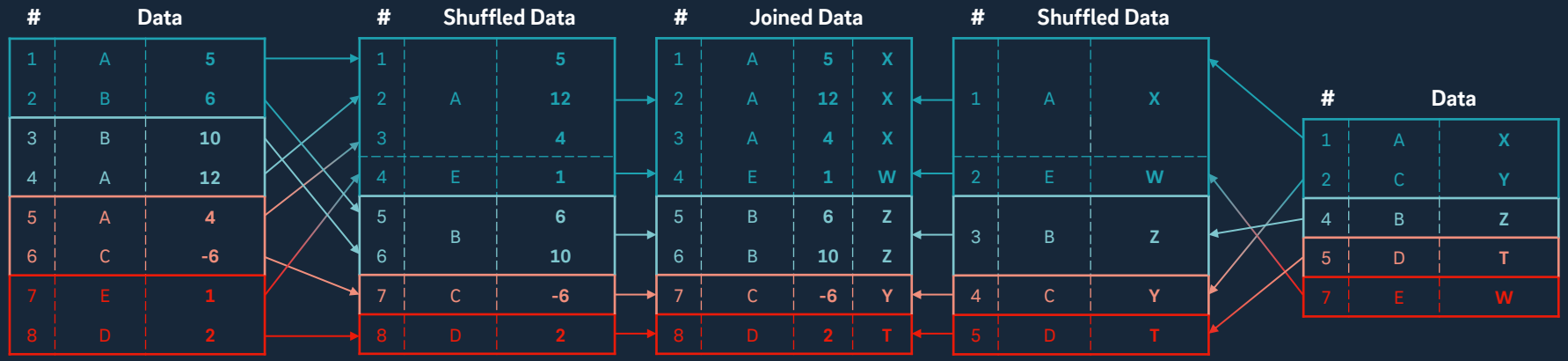


Operația de grupare și transformare - Window

Când Spark SQL utilizează expresii și nu funcții definite de dezvoltator, codul de agregare este generat direct în limbajul nativ Spark, sporind eficiența. Operația de grupare în sine se folosește de repartiționarea din Spark Core.

Data Frame -> Map-Reduce – Operațiile de asociere

Operația de asociere din Spark SQL este în spate efectuată folosind operația de reduce din Map-Reduce, aplicată pe fiecare set de date în parte, iar apoi folosind operația de mapare pe perechile de partiții care conțin aceeași chei.



Operația de asociere - JOIN

Când Spark SQL utilizează expresii și nu funcții definite de dezvoltator, codul de join este generat direct în limbajul nativ Spark, sporind eficiența. Ambele seturi de date vor fi repartiționate în același număr de partiții și cheile plasate similar.

Data Frame -> RDD

Fiind o extensie a Spark Core, și implementând operații care pot fi translatate în operații de Map-Reduce, Spark SQL permite ca lanțul de operații al unui Data Frame să fie convertit într-un lanț de comenzi RDD direct.

➢ Crearea și accesarea unui obiect de tip RDD dintr-un Data Frame

```
data_rdd = data_df.rdd
```

❖ Un nou obiect de tip RDD este returnat format din lanțul de comenzi al Data Frame-ului. Obiectele din RDD vor fi de tip-ul intern Row. Data Frame-ul de la care a pornit proiecția va rămâne neschimbat.

```
print(data_rdd.collect())
```

```
[
 Row(nume='Vali', varsta=23, ocupatie='Programator', vechime=4, inactiv=None, zona='A', extra=['3D Printer', 'XBOX']),
 Row(nume='Vlad', varsta=34, ocupatie='Instalator', vechime=11, inactiv=None, zona='B', extra=['EV']),
 Row(nume='Bea', varsta=29, ocupatie='Reporter', vechime=7, inactiv=True, zona='B', extra=None)
]
```

⚠ **Orice cod generat de Spark SQL în limbajul nativ Spark va fi convertit în metode Python, reducând eficiența operațiilor.**

RDD -> Data Frame

Pe de altă parte, lanțul de comenzi a unui RDD nu poate fi întotdeauna convertit la un lanț de comenzi Data Frame întrucât nu toate operațiile aplicabile pe RDD-uri pot fi translatate și nu este garantată o schemă a datelor, dar datele după rularea lanțului de comenzi al RDD-ului pot fi folosite pentru crearea unui Data Frame nou.

- Crearea unui Data Frame prin parsarea datelor și validarea schemei din datele unui RDD

```
data_df = spark.createDataFrame(data_rdd, schema=['nume', 'varsta', 'ocupatie', 'vechime', 'inactiv', 'zona', 'extra'])
```

- Este returnat un nou Data Frame, format prin parsarea / validarea fiecărei partiție a RDD-ului și păstrarea lor în exact aceeași configurație. RDD-ul de la care a pornit proiecția va rămâne neschimbat.

```
data_df.show()
```

```
+----+------+-----------+-------+-------+----+------------------+
|nume|varsta|   ocupatie|vechime|inactiv|zona|             extra|
+----+------+-----------+-------+-------+----+------------------+
|Vali|    23|Programator|      4|   NULL|   A|[3D Printer, XBOX]|
|Vlad|    34| Instalator|     11|   NULL|   B|              [EV]|
| Bea|    29|   Reporter|      7|   true|   B|              NULL|
+----+------+-----------+-------+-------+----+------------------+
```

Spark ML

Not one for all, but all for one

Expert
Services

PySpark ML

PySpark ML este librăria de Apache Spark ML pentru Python. Este foarte similară cu populara librărie sklearn, oferind o structură similară de apelare împărțită în două etape, etapa opțională de antenare și cea de predicție. A fost concepută pentru a oferi oricărei persoane capabilitatea de a aplic algoritmi de învățare automată pe date.

PySpark ML oferă:

- Un tip de coloană nou, pentru a lucra cu atributele datelor, numită Vector.
- O clasă pentru construirea coloanelor de tip Vector.
- Diverse clase pentru standardizarea datelor.
- Diverse clase pentru modele de învățare automată.

Codul PySpark ML este unul foarte simplu, întrucât complexitatea este în ce facem cu datele.

Modele de Învățare Automată

PySpark ML oferă clase de Python care implementează modele cunoscute de învățare automată. Fiecare model are parametrii săi unici, dar toate implementează o interfață de apelare comună care se regăsește și în librăria sklearn.

```
class MLModel:
   def __init__(self, val1=None , val2=2):

   def setVal1(self, new_value):

   def setVal2(self, new_value):

   ...

   def fit(self, data_df):
```

Interfața unui model care necesită antrenare pe date.

Parametrii modelor care necesită antrenare din Spark ML pot fi specificați fie la inițializare, fie prin utilizarea metodelor dedicate de configurare, întrucât modelele adoptă șablonul de builder.

```
model = MLModel().setVal1(3).setVal2('input')
```

```
model = MLModel(val1=3).setVal2('input')
```

Modelele de învățare automată implică o fază de antrenare pe date. Pentru acest scop este oferită funcția fit. Aceasta va antrena modelul pe date și va returna un model antenat.

Modele de Transformare

Modelele antrenate și modele care nu necesită o etapă de antrenare implementează și ele o interfață de apelare comună, care se regăsește și în librăria sklearn, în general utilizată pentru transformarea ulterioară a datelor.

Interfața unui model antrenat / care nu necesită antrenare.

```
class PySparkMLModelInterface:
    def __init__(self, val1=None , val2=2):

    def setVal1(self, new_value):

    ...


    def transform(self, data_df):


    def save(self, path):

    def load(self, path):
```

Parametrii modelor care nu necesită antrenare din Spark ML pot fi specificați fie la inițializare, fie prin utilizarea metodelor dedicate de configurare, întrucât modelele adoptă șablonul de builder.

```
model = MLModel(val1=3).setVal2('input')
```

Pentru rularea modelului pe date, există funcția transform. Aceasta va aplica operațiile necesare rulării modelului pe Data Frame-ul furnizat și va returna un nou Data Frame.

Toate modelele includ metoda save pentru stocarea modelului pe disk și metoda load pentru încărcarea unui model salvat anterior.

Transformarea datelor

Majoritatea algoritmilor de învățare automată pot lucra numai pe date numerice. Pentru a extrage o informație numerică din o coloană text, Spark ML oferă un model care convertește valorile text în funcție de frecvența lor în date.

- Antrenarea modelului care extrage informația numerică dintr-o coloană text

```
from pyspark.ml.feature import StringIndexer

string_indexer = StringIndexer(inputCol='ocupatie', outputCol='ocupatie_idx', stringOrderType='frequencyDesc')
trained_string_indexer = string_indexer.fit(data_df)
```

  - Un nou model antrenat este returnat care poate asigna valorilor din coloana „ocupatie” o valoare în noua coloană „ocupatie_idx” bazat pe frecvența valorii în datele de antrenament, începând de la 0.

Mai multe informații privind parametrii și comportamentul modelului se găsesc la: https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.feature.StringIndexer.html.

Conversia textului: StringIndexer – Antrenarea Modelului

Majoritatea algoritmilor de învățare automată pot lucra numai pe date numerice. Pentru a extrage o informație numerică din o coloană text, Spark ML oferă un model care convertește valorile text în funcție de frecvența lor în date.

- Antrenarea modelului care extrage informația numerică dintr-o coloană text

```
from pyspark.ml.feature import StringIndexer

string_indexer = StringIndexer(inputCol='ocupatie', outputCol='ocupatie_idx', stringOrderType='frequencyDesc')
trained_string_indexer = string_indexer.fit(data_df)
```

  - Un nou model antrenat este returnat care poate asigna valorilor din coloana „ocupatie” o valoare în noua coloană „ocupatie_idx” bazat pe frecvența valorii în datele de antrenament, începând de la 0.

Mai multe informații privind parametrii și comportamentul modelului se găsesc la: https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.feature.StringIndexer.html.

Conversia textului: StringIndexer – Transformarea datelor

Modelul antrenat anterior poate fi folosit ori pe aceleași date, ori pe alte date de test pentru a transforma coloana text într-o coloană de tip numeric pentru a fi folosită ulterior de alte modele de învățare automată.

- Extragerea de informație numerică dintr-o coloană text folosind modelul antrenat.

```
data_string_indexed_df = trained_string_indexer.transform(data_df)
```

- Precum metodele Data Frame-urilor, un nou obiect de tip Data Frame este returnat, în acest caz având coloana „ocupatie_idx" adăugată sau actualizată cu o valoare bazată pe frecvența valorii în date.

```
transformed_data_df.show()
```

```
+----+------+-----------+-----------+-------+-------+----+-----------------+
|nume|varsta|   ocupatie|ocupatie_idx|vechime|inactiv|zona|            extra|
+----+------+-----------+-----------+-------+-------+----+-----------------+
|Vali|    23|Programator|        0.0|      4|   NULL|   A|[3D Printer, XBOX]|
|Vlad|    34| Instalator|        1.0|     11|   NULL|   B|             [EV]|
| Bea|    29|   Reporter|        2.0|      7|   true|   B|             NULL|
+----+------+-----------+-----------+-------+-------+----+-----------------+
```

Tipul de date și coloană Vector

Anumite modele PySpark ML, pentru a eficientiza memoria folosită în timpul antrenării / rulării, folosesc un tip special de date, Vector. Ea reprezintă o listă de valori numerice (vector în mai multe dimensiuni) reținută eficient.

PySpark oferă funcții speciale pentru a inițializa o variabilă vector. Toate se află în același clasă de Python.

```
from pyspark.ml.linalb import Vectors
```

- ❑ Vector Dens – Un vector standard în n dimensiuni

```
vector = Vectors.dense(0.0, 5.0, 7.0)
```

➢ Vector (0.0, 5.0, 7.0)

- ❑ Vector Rar – Un vector care doar elementele non-zero, economisind memorie când majoritatea valorilor sunt zero.

```
vector = Vectors.sparse(5, 0.0, 5.0, 7.0, 0.0, 0.0)
```

➢ Vector Rar (0.0, 5.0, 7.0, 0.0, 0.0)

✓ Unele modele folosesc intern acest tip de date pentru a reține coeficienții modelelor antrenate. Se pot construi și coloane de Data Frame cu acest tip de date pentru a fi folosit sau returnat de anumite modele din PySpark ML.

Conversia textului: CountVectorizer – Antrenarea Modelului

Datorită eficienței tipului Vector, modelele care calculează liste de valori preferă acest format pentru returnare. Un astfel de model este cel care determină frecvența fiecărei valori unice într-o coloană.

- Antrenarea modelului care extrage numărul de apariții a unui valori dintr-o listă de valori

```
from pyspark.ml.feature import CountVectorizer

count_vectorizer = CountVectorizer(inputCol='extra', outputCol='consumatori', vocabSize=3)
trained_count_vectorizer = count_vectorizer.fit(data_df.withColumn('extra', f.coalesce('extra', f.array())))
```

  - Un nou model antrenat este returnat care poate contorizează numărul de apariții a fiecărei valori din coloana „extra” (nu se acceptă NULL) și construiește o nouă coloană „consumatori” de tip Vector cu aceste valori.

Mai multe informații privind parametrii și comportamentul modelului se găsesc la:
https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.feature.CountVectorizer.html .

Conversia textului: CountVectorizer – Transformarea datelor

Modelul antrenat anterior poate fi folosit ori pe aceleași date, ori pe alte date de test pentru a transforma coloana extra într-o coloană de tip vector pe conținând numărul de apariției fiecărei valori.

- Extragerea numărului de apariții a unui valori dintr-o listă de valori folosind modelul antrenat

```
data_string_indexed_df = data_string_indexed_df.withColumn('extra', f.coalesce('extra', f.array()))
data_with_value_counts_df = trained_count_vectorizer.transform(data_string_indexed_df)
```

- Precum metodele Data Frame-urilor, un nou obiect de tip Data Frame este returnat, în acest caz având coloana „consumatori” adăugată sau actualizată, de tip Vector Rar, cu o valoare bazată pe frecvența valorii în date.

```
data_with_value_counts_df.show()
```

```
+----+------+-----------+-----------+-------+-------+----+-----------------+-------------------+
|nume|varsta|   ocupatie|ocupatie_idx|vechime|inactiv|zona|            extra|         consumatori|
+----+------+-----------+-----------+-------+-------+----+-----------------+-------------------+
|Vali|    23|Programator|        0.0|      4|   NULL|   A|[3D Printer, XBOX]|(3,[0,2],[1.0,1.0])|
|Vlad|    34| Instalator|        1.0|     11|   NULL|   B|             [EV]|      (3,[1],[1.0])|
| Bea|    29|   Reporter|        2.0|      7|   true|   B|               []|          (3,[],[])|
+----+------+-----------+-----------+-------+-------+----+----+------------+-------------------+
```

Pregătirea datelor de intrare: VectorAssembler

Pentru a crea o coloană de tipul Vector, PySpark ML oferă modele de transformare pentru acest scop. Coloana este folosită ulterior de alte modele de învățare automată care necesită acest tip de date la intrare.

- Un model pentru concatenarea coloanele existente într-un Vector pentru a fi folosit de modele ulterioare

```
from pyspark.ml.feature import VectorAssembler

vector_assembler = VectorAssembler(inputCols=['varsta', 'ocupatie_idx', 'consumatori'], outputCol='atribute')
data_vectorized_df = vector_assembler.transform(data_with_value_counts_df)
```

- Un nou obiect de tip Data Frame este returnat care are adăugată sau actualizată coloana „atribute”, de tip Vector, care va reține o listă formată din valorile coloanelor „varsta”, „ocupatie_idx” și „vechime”.

```
data_vectorized_df.show()
```

```
+----+------+----------+-------------+-------+-------+----+----------------+------------------+----------------------+
|nume|varsta|   ocupatie|ocupatie_idx|vechime|inactiv|zona|           extra|        consumatori|              atribute|
+----+------+----------+-------------+-------+-------+----+----------------+------------------+----------------------+
|Vali|    23|Programator|          0.0|      4|   NULL|   A|[3D Printer, XBOX]|(3,[0,2],[1.0,1.0])|[23.0,1.0,1.0,0.0,1.0]|
|Vlad|    34| Instalator|          1.0|     11|   NULL|   B|            [EV]|      (3,[1],[1.0])|[34.0,0.0,0.0,1.0,0.0]|
| Bea|    29|   Reporter|          2.0|      7|   true|   B|              []|          (3,[],[])|[29.0,2.0,0.0,0.0,0.0]|
+----+------+----------+-------------+-------+-------+----+----------------+------------------+----------------------+
```

Conversia textului: KMeans – Antrenarea Modelului

Cele mai multe modele de învățare automată necesită la intrare o coloană de tip Vector, fiind proiectate să lucreze cu date multidimensionale, și, dacă este cazul, returnează rezultatul la ieșire o coloană de același tip.

- Antrenarea modelului de învățare automată de clusterizare KMeans

```
from pyspark.ml.clustering import KMeans

kmeans = KMeans(featuresCol='atribute', predictionCol='cluster', maxIter=10, k=2)
trained_kmeans = kmeans.fit(data_vectorized_df)
```

  - Un nou model antrenat de KMeans este returnat care pe baza clusterelor sale poate asigna, pe baza valorilor din coloana „atribute”, indicele celui mai apropriat cluster într-o nouă coloană „cluster”.

Mai multe informații privind parametrii și comportamentul modelului se găsesc la:
https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.clustering.KMeans.html .

Conversia textului: KMeans – Transformarea datelor

Modelul antrenat anterior poate fi folosit ori pe aceleași date, ori pe alte date de test pentru a calcula folosind valorile din coloana „atribute" indicele celui mai apropriat dintre clusterele calculate la pasul de antrenare din date.

- Asignarea datelor de intrare la un cluster folosind modelul antrenat

```
final_data_df = trained_kmeans.transform(data_vectorized_df)
```

- Precum metodele Data Frame-urilor, un nou obiect de tip Data Frame este returnat, în acest caz având coloana „cluster" adăugată sau actualizată cu indicele clusterului la care rândul din date a fost asignat.

```
final_data_df.show()
```

```
+----+------+-----------+------------+-------+-------+----+----------------+------------------+----------------------+-------+
|nume|varsta|   ocupatie|ocupatie_idx|vechime|inactiv|zona|           extra|        consumatori|              atribute|cluster|
+----+------+-----------+------------+-------+-------+----+----------------+------------------+----------------------+-------+
|Vali|    23|Programator|         0.0|      4|   NULL|   A|[3D Printer, XBOX]|(3,[0,2],[1.0,1.0])|[23.0,1.0,1.0,0.0,1.0]|      1|
|Vlad|    34| Instalator|         1.0|     11|   NULL|   B|            [EV]|      (3,[1],[1.0])|  (5,[0,3],[34.0,1.0])|      0|
| Bea|    29|   Reporter|         2.0|      7|   true|   B|              []|          (3,[],[])|  (5,[0,1],[29.0,2.0])|      0|
+----+------+-----------+------------+-------+-------+----+----+-------------+------------------+----------------------+-------+
```

Salvarea și Încărcarea Modelelor de pe disk

Toate modelele antrenate precum și cele care nu necesită etapa de antrenare pot fi salvate și încărcate de pe disk. În general, salvam modelele după pasul de antrenare și le încărcam de pe disk la pasul de inferare a datelor.

- Salvarea modelele antrenate până în acest punct

```
trained_string_indexer.save(save_path)
trained_count_vectorizer.save(save_path)
```

```
vector_assembler.save(save_path)
trained_kmean.save(save_path)
```

  - Modelele nu pot fi salvate decât într-un folder gol

- Încărcarea modelelor salvate de pe disk

```
trained_string_indexer = StringIndexer.load(save_path)
trained_count_vectorizer = CountVectorizer.load(save_path)
```

```
vector_assembler = VectorAssembler.load(save_path)
trained_kmean = Kmeans.load(save_path)
```

  - Modelele încărcate pot fi rulate pe date în continuare ca normal.

Mai multe modele și detalii lor se găsesc in documentația Spark ML: https://spark.apache.org/docs/latest/ml-guide.html

Un program PySpark ML

```
from pyspark.sql import SparkSession, functions as f
from pyspark.ml import feature as mlf, clustering as mlc

spark = SparkSession.builder.master('local[*]').getOrCreate()
data_df = spark.read.format('json').load('/path/to/course/data/folder')
data_df = data_df.withColumn('extra', f.coalesce('extra', f.array()))

string_indexer = mlf.StringIndexer(inputCol='ocupatie', outputCol='ocupatie_idx', stringOrderType='frequencyDesc')
data_string_indexed_df = string_indexer.fit(data_df).transform(data_df)

count_vectorizer = mlf.CountVectorizer(inputCol='extra', outputCol='consumatori', vocabSize=3)
data_with_value_counts_df = count_vectorizer.fit(data_df).transform(data_string_indexed_df)

vector_assembler = mlf.VectorAssembler(inputCols=['varsta', 'ocupatie_idx', 'consumatori'], outputCol='atribute')
data_vectorized_df = vector_assembler.transform(data_with_value_counts_df)

kmeans = mlc.KMeans(featuresCol='atribute', predictionCol='cluster', maxIter=10, k=2)
clustered_data_df = kmeans.fit(data_vectorized_df).transform(data_vectorized_df)
clustered_data_df.write.format('json').save('/path/to/save/folder')
```