



# Expert Service

Hands On Advanced Analytics with Apache Spark

Curs II

# Apache Spark

Not the 🔥 spark



Expert  
Services

# Până în acest punct – Schema datelor

➤ Data Frame: Tabel de Date, fiecare coloană conține același tip de date

➤ Principalele tipuri de date din Spark

❖ integer    ❖ string    ❖ double    ❖ date    ❖ map<tip cheie, tip valoare>  
❖ boolean    ❖ array<tip element>    ❖ float    ❖ timestamp    ❖ struct<coloana1:tip 1, coloana2:tip 2,...>

➤ Definiția tipurilor de date folosind obiectele PySpark

```
from pyspark.sql import types as T
data_schema = T.StructType([
    T.StructField('nume', T.StringType(), False),
    T.StructField('varsta', T.IntegerType(), False),
    T.StructField('ocupatie', T.StringType(), False),
    T.StructField('vechime', T.IntegerType(), False),
    T.StructField('inactiv', T.BooleanType(), True),
    T.StructField('zona', T.StringType(), True),
    T.StructField('extra', T.ArrayType(T.StringType()), True)
])
```

# Până în acest punct – Sesiune Spark și Citire

- Sesiune Spark: Poarta de acces către Spark SQL

```
spark = SparkSession.builder.getOrCreate()
```

- Crearea unui Data Frame Spark dintr-o listă de Python.

```
data_df = spark.createDataFrame(data, schema=data_schema)
```

- Citirea fișierelor de tip CSV / JSON / Parquet într-un Data Frame

```
data_df = spark.read.format('csv').option('header', 'true').schema(data_schema).load('/path/to/folder/or/file')
```

```
data_df = spark.read.format('json').schema(data_schema).load('/path/to/folder/or/file')
```

```
data_df = spark.read.format('parquet').schema(data_schema).load('/path/to/folder/or/file')
```

# Până în acest punct – Afișare, Colectare și Scriere

- Afișare tipurilor de date la consolă

```
data_df.printSchema()
```

- Colectarea datelor într-o listă de Python

```
data_list = data_df.collect()
```

- Afișare datelor la consolă

```
data_df.show()
```

- Colectarea datelor într-o tabelă de Pandas

```
data_pdf = data_df.toPandas()
```

- Scrierea datelor în fișiere de tip CSV / JSON / Parquet

```
data_df.write.format('csv').option('header', 'true').save('/path/to/save/folder')
```

```
data_df.write.format('json').save('/path/to/save/folder')
```

```
data_df.write.format('parquet').save('/path/to/save/folder')
```

# Până în acest punct – The „Hello World” of PySpark

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

```
spark = SparkSession.builder.getOrCreate()
```

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

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

- Pas 3: Afișare tipurilor de date la consolă

```
data_df.printSchema()
```

- Pas 4: Afișare datelor la consolă

```
data_df.show()
```

- Pas 5: Scrierea datelor dintr-un Data Frame într-un folder

```
data_df.write.format('json').save('/path/to/save/folder')
```

# Înlănțuirea metodelor

Ce face Spark atât de simplu, dar în același timp și atât de diferit la prima vedere, este înlănțuirea metodelor.

## ➤ Exemplu 1: Diverse configurări a Sesiunii Spark

```
spark = SparkSession.builder.master('local[*]').config('spark.driver.memory', '3g').appName('Lab2').getOrCreate()
```

Apelarea funcțiilor de schimbare a opțiunilor implicite.

## ➤ Exemplu 2: Diverse configurări a citirii datelor Spark

```
data_df = spark.read.format('json').option('multiLine', 'true').schema(data_schema).load('/path/to/folder/or/file')
```

Apelarea funcțiilor de schimbare a opțiunilor implicite.

Această înlănțuire ne permite să specificăm mulți parametrii într-un mod elegant și ușor de citit.

# Înlănțuirea multor metode

Prin înlănțuirea multor metode și funcții putem ajunge la linii foarte mari de cod. Pentru a evita acest lucru, se recomandă împărțirea lor pe mai multe rânduri. În Python, putem face asta prin folosirea parantezelor, ( și ).

## ➤ Diverse configurări a Sesiunii Spark

```
spark = (  
    SparkSession.builder  
        .master('local[*]')  
        .config('spark.driver.host', 'localhost')  
        .config('spark.driver.memory', '3g')  
        .config('spark.executor.host', 'localhost')  
        .appName('HandsOnLab2')  
        .getOrCreate()  
)
```

Apelarea funcțiilor de schimbare a opțiunilor. Chiar dacă nu le specificăm, Spark în spate are valori implicite pentru toate opțiunile. Ele sunt documentate la: <https://spark.apache.org/docs/latest/configuration.html>

Această tehnică de înlănțuire provine din Java Builder, un șablon foarte popular pentru Java, limbajul de origine Spark.



# Java Builder

În programarea orientată obiect (OOP), design pattern-urile sunt șabloane pentru a rezolva probleme comune de design a software-ului. Unul dintre cele mai cunoscute design pattern-uri este Builder, provenit din limbajul Java.

```
class CarBuilder:  
    def __init__(self):  
        self.car = {'wheels': 4, 'color': 'Silver'}
```

```
    def set_wheels(self, number):  
        self.car['wheels'] = number  
        return self
```

```
    def set_color(self, color):  
        self.car['color'] = color  
        return self
```

```
    def build(self):  
        return self.car
```

## Implementarea șablonului și în Python.

Inițializarea builder-ului cu valori implicite potrivite.

Funcțiile de setare a opțiunilor. Șablonul de builder se remarcă prin returnarea obiectul de `self` la metodele sale de setare. Asta permite înlănțuirea metodelor:

```
car_builder = CarBuilder().set_color('Red').set_wheels(4)
```

Funcțiile de finalizare, care returnează rezultatul

```
car = CarBuilder().set_color('Red').set_wheels(4).build()
```

# Java Builder – Utilizare in PySpark

Având la bază limbajul Scala, bazat pe Java, și fiind și un Framework bazat pe construcții de transformări, Spark, și respectiv PySpark, se folosesc de acest șablonul de programare Builder în API-ul pe care îl oferă. Am întâlnit până acum aceste construcții:

```
spark = SparkSession.builder.master('local[*]').config('spark.driver.memory', '3g').getOrCreate()
```

Inițializarea builder-ului  
cu valori implicite  
potrivite alese de  
PySpark.

Apelarea funcțiilor de schimbare a  
opțiunilor implicite.

Apelarea funcției de finalizare  
care returnează obiectul  
construit Spark ↑ / Data Frame ↓

```
data_df = spark.read.format('csv').option('header', 'true').schema(data_schema).load('/path/to/folder/or/file')
```

❖ Similar și la scriere.

```
data_df.write.format('csv').option('header', 'true').save('/path/to/folder/or/file')
```

# Comenzi de transformare

Ce nu trebuie, nu facem



## Expert Services



# Date de Test

Pentru a testa cod de Spark, în general, se folosesc liste mici cu date de test la crearea unui Data Frame, în loc să citim datele. Când testăm aplicații de Spark, putem folosi această metodă pentru scrierea de test.

- Funcția `createDataFrame` ne permite să specificăm doar numele coloanelor, fără a fi nevoie să specificăm și tipul lor. Spark va trece prin data și va încerca să deducă automat tipul de date.

```
data = [  
    ['Vali', 23, 'Programator', 4, None, 'A', ['3D Printer', 'XBOX']],  
    ['Vlad', 34, 'Instalator', 11, None, 'B', ['EV']],  
    ['Bea', 29, 'Reporter', 7, True, 'B', None]  
]
```

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

- ❖ Aceste date de test sunt folosite de acum încolo pentru exemplificarea metodelor de transformare. Ele sunt stocate și în format JSON sau Parquet în setul de date de exemplu pentru acest curs.

# Proiectarea Datelor - Select

Data Frame-urile oferă și metode pentru proiectarea datelor, pe lângă cele de scriere, colectare și afișare, similare cu interogările din bazele de date SQL, în același stil de apelare prin înlănțuire.

- Operația de Selecție – Păstrarea doar anumite coloane

```
new_data_df = data_df.select('nume', 'varsta', 'extra')
```

- ❖ Un nou obiect de tip Data Frame este returnat care are doar coloanele nume, varsta și extra. Data Frame-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

new\_data\_df.show()

nume	varsta	extra
Vali	23	[3D Printer, XBOX]
Vlad	34	[EV]
Bea	29	NULL

# Proiectarea Datelor - Ștergere

De multe ori se întâmplă să lucrăm cu date care conțin foarte multe coloane, și devine inpractic să le selectăm. În loc de selecția coloanelor pe care dorim să le păstrăm, putem scoate coloanele de care nu avem nevoie.

➤ Operația de Ștergere – Ștergere a coloanelor

```
new_data_df = data_df.drop('varsta', 'zona', 'extra')
```

❖ Un nou obiect de tip Data Frame este returnat care nu mai are coloanele varsta, concediu și extra. Data Frame-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

new\_data\_df.show()

nume	ocupatie	inactiv	vechime
Vali	Programator	NULL	4
Vlad	Instalator	NULL	11
Bea	Reporter	true	7

# Proiectarea Datelor – Redenumirea Coloanelor

Numele coloanele din date adesea nu se potrivesc cu metodologia noastră de a numi coloanele, de exemplu conțin caractere neobișnuite sau sunt în altă limbă. Putem să redenumim aceste coloane pentru a le standardiza.

## ➤ Operația de Redenumire – Redenumire a coloanelor

```
new_data_df = data_df.withColumnRenamed('ocupatie', 'job')
```

- ❖ Un nou obiect de tip Data Frame este returnat care nu mai are coloana post, dar are o nouă coloană job cu aceleași valori. Data Frame-ul de la care a pornit proiecția va rămâne neschimbat.

```
data_df.printSchema()
```

```
root
 |-- nume: string (nullable = false)
 |-- varsta: integer (nullable = false)
 |-- ocupatie: string (nullable = false)
 |-- vechime: integer (nullable = true)
 |-- inactiv: boolean (nullable = true)
 |-- zona: string (nullable = false)
 |-- extra: array (nullable = true)
 |     |-- element: string (containsNull = true)
```

```
new_data_df.printSchema()
```

```
root
 |-- nume: string (nullable = false)
 |-- varsta: integer (nullable = false)
 |-- job: string (nullable = false)
 |-- vechime: integer (nullable = true)
 |-- inactiv: boolean (nullable = true)
 |-- zona: string (nullable = false)
 |-- extra: array (nullable = true)
 |     |-- element: string (containsNull = true)
```

# Un program simplu PySpark

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

```
from pyspark.sql import SparkSession  
spark = SparkSession.builder.getOrCreate()
```

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

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

- Pas 3: Ștergerea colanelor cu date personale

```
data_without_personal_columns_df = data_df.drop('varsta', 'zona', 'extra')
```

- Pas 4: Redenumirea coloanei de post

```
final_data_df = data_without_personal_columns_df.withColumnRenamed('ocupatie', 'job')
```

- Pas 5: Scrierea datelor finale într-un folder

```
final_data_df.show()
```



# Data Frame – View și Plan de Execuție

Un Data Frame 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 transformările care trebuie efectuate, numite Planul Logic de Execuție.



Citirea Datelor din `/path/to/course/data/folder`

Ștergerea Coloanelor `varsta`, `zona` și `extra`

Redenumirea Coloanei `ocupatie` în `job`

O vedere internă a unui Data  
Frame

Planul Logic de Execuție

Calcululele 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 Data Frame-ului, la executori, care îl vor efectua.



Citiți Datele din `/path/to/course/data/folder`

Ștergeți Coloanele `varsta`, `zona` și `extra`

Redenumiți Coloana `ocupatie` în `job`

Scrieți datele la `/path/to/save/folder`

Planul Fizic de Execuție



Master Machine



Spark Driver



Spark Context



Executor Machines

# Data Frame – Imutabilitate

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



Citirea Datelor din `/path/to/course/data/folder`  
Ștergerea Coloanelor `varsta`, `zona` și `extra`

Data Frame-ul inițial



Citirea Datelor din `/path/to/course/data/folder`  
Ștergerea Coloanelor `varsta`, `zona` și `extra`  
Redenumirea Coloanei `ocupatie` în `job`

Data Frame-ul după o operație de transformare

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

Această proprietate se numește în programare proprietatea de imutabilitate.

# Un program simplu PySpark – Planul de Execuție

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



Citirea Datelor din `/path/to/course/data/folder`

Data Frame: `data_df`



```
data_without_personal_columns_df = data_df.drop('varsta', 'extra')
```



Citirea Datelor din `/path/to/course/data/folder`  
Ștergerea Coloanelor `varsta`, `zona` și `extra`

Data Frame: `data_without_personal_columns_df`



```
final_data_df = data_without_personal_columns_df.withColumnsRenamed('post', 'job')
```



Citirea Datelor din `/path/to/course/data/folder`  
Ștergerea Coloanelor `varsta`, `zona` și `extra`  
Redenumirea Coloanei `post` în `job`

Data Frame: `final_data_df`

# Planul de Execuție - Optimizări

O funcționalitate de baza din Spark SQL este procesul de optimizare a operațiilor, Optimizatorul Catalyst. El preia lanțul de comenzi dintr-un Data Frame și încearcă să aplice optimizări pertinente care pot îmbunătății timpii de execuție și memoria folosită de executori.



Citirea Datelor din `/path/to/course/data/folder`  
Ștergerea Coloanelor `varsta`, `zona` și `extra`  
Redenumirea Coloanei `ocupatie` în `job`

Planul Logic de Execuție: `final_data_df`

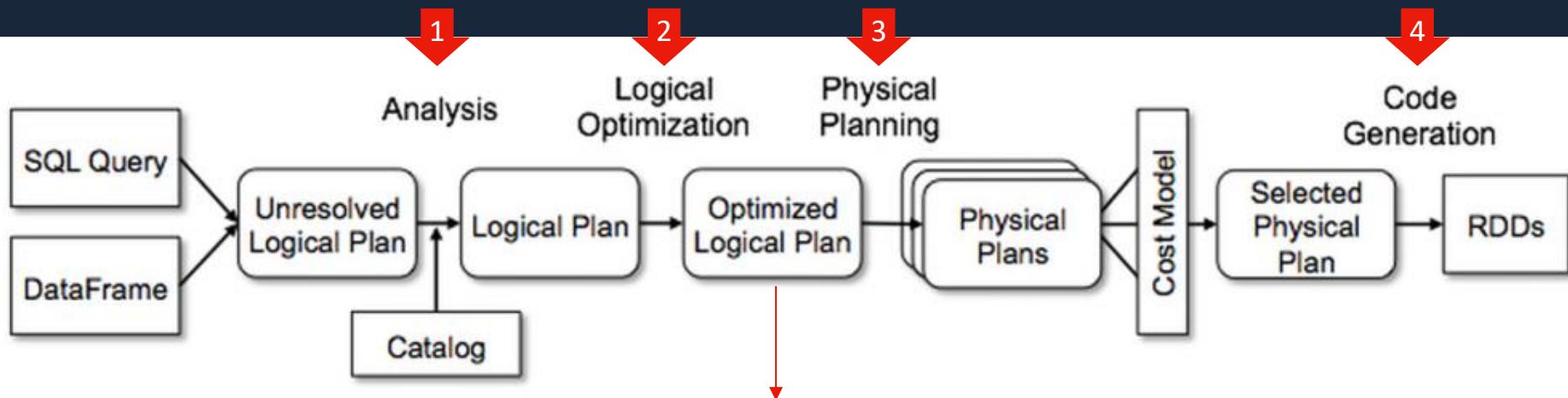


Citirea Coloanelor `nume`, `ocupatie`, `vechime`, `concediu` din `/path/to/course/data/folder`  
Redenumirea Coloanei `ocupatie` în `job`

Planul Logic Optimizat: `final_data_df`

Optimizatorul Catalyst este unul dintre principalele motive pentru care Spark este atât de rapid și ușor de utilizat.

# Optimizatorul Catalyst



## Optimizări Logice:

- Filtrare la sursa
- Comparații text

## Optimizări Fizice:

- Serializarea datelor
- Sort înainte de Join

# Data Frame – Afișarea Planului de Execuție

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

➤ Pentru a printa toate planurile de execuție, logice și fizice, la consolă, folosim:

```
final_data_df.explain('extended')

== Parsed Logical Plan ==
Project [inactiv#86, nume#88, ocupatie#89 AS job#103, vechime#91L]
+- Project [inactiv#86, nume#88, ocupatie#89, vechime#91L]
   +- Relation [inactiv#86,extra#87,zona#92,nume#88,ocupatie#89,varsta#90L,vechime#91L]

...
```

Mai multe detalii găsiți la <https://spark.apache.org/docs/latest/sql-ref-syntax-qry-explain.html>

# Înlănțuirea Comenzilor

Întrucât toate funcțiile de transformare din PySpark vor returna un obiect nou de tip Data Frame, asta ne permite foarte ușor să înlănțuim codul de transformare a datelor.

- Operația de Ștergere și Operația de Redenumire a Coloanelor scrise separat

```
data_without_personal_columns_df = data_df.drop('varsta', 'zona', 'extra')
final_data_df = data_without_personal_columns_df.withColumnsRenamed('ocupatie', 'job')
```

- Operația de Ștergere și Operația de Redenumire a Coloanelor scrise împreună.

```
final_data_df = (
    data_df
    .drop('varsta', 'zona', 'extra')
    .withColumnsRenamed('ocupatie', 'job')
)
```

# Metode și expresii de transformare

Încă nu ai curățat datele?



Expert  
Services



# Comenzi de transformare, filtrare și ordonare

Spark poate transmite și comenzi mai complexe, comenzi de transformare, filtrare sau ordonare, către executori. Aceste comenzi permit calcularea unor noi valori din datele existente prin utilizarea de expresii matematice, asemenea bazelor de date sau formulelor de Excel.



Citirea Datelor din `/path/to/course/data/folder`

⊕ Adăugarea Coloanei `varsta_de_angajare = varsta - vechime`

📌 Păstrarea Datelor îndeplinesc condiția `varsta < 30`

Data Frame

- ⊕ Comenzile de transformare: pe baza unei expresii se calculează coloane noi sau se le înlocuiesc cele existente.
- 📌 Comenzile de filtrare și ordonare: pe baza unei expresii se decide care date se păstrează și ordinea lor.

# Column - Expresii de calcul

Pentru a reprezenta și construi expresii de calcul programatic, care vor fi adăugate ca noi coloane sau vor înlocui coloanele existente, PySpark folosește o clasă specială de Python cu un nume intuitiv, numită `Column`. Ea reține doar „formula” care trebuie efectuată, similar cum un Data Frame reține doar lanțul de comenzi, pentru noile coloane.

PySpark oferă funcții speciale pentru a construi astfel de expresii. Toate se află în același modul de Python.

```
from pyspark.sql import functions as f
```

➤ O expresie ce reprezintă o valoare constantă

```
expr = f.lit(130)
```

➤ O expresie ce reprezintă valoarea unei coloane

```
expr = f.col('varsta')
```

➤ Afișarea expresiei de calcul, definiția coloanei:

```
print(expr)
```

```
Column<'130'>
```

```
print(expr)
```

```
Column<'varsta'>
```

# Column - Expresii de calcul

Daca **NU** sunt transformari/functii utilizate, atunci numele coloanelor pot fi apelate ca strings .

```
final_data_df = (  
    data_df  
    .select('varsta', 'zona', 'extra')  
)
```


```
final_data_df = (  
    data_df  
    .groupBy('zona')  
    .count()  
)
```

Cand aplicam **transformari/functii**, atunci este necesara

- specificarea 'column type' utilizand **col**
- convertirea valorilor constante in 'column type' utilizand **lit**


```
from pyspark.sql.functions import col, lit, upper
```

```
final_data_df = (  
    data_df  
    .select(upper(col('varsta')), upper(col('zona')), upper(col('extra')))  
)
```



```
from pyspark.sql.functions import col, lit, concat
```

```
final_data_df = (  
    data_df  
    .select(concat(col('vechime'), lit(' ani') ) )  
)
```



# Column - Expresii de calcul

Daca **NU** sunt transformari/functii utilizate, atunci numele coloanelor pot fi apelate ca strings .


```
final_data_df = (  
    data_df  
    .select('varsta', 'zona', 'extra')  
)
```

```
final_data_df = (  
    data_df  
    .groupBy('zona')  
    .count()  
)
```


Cand aplicam **transformari/functii**, atunci este necesara

- specificarea 'column type' utilizand **col**
- convertirea valorilor constante in 'column type' utilizand **lit**

```
from pyspark.sql.functions import col, lit, upper  
  
final_data_df = (  
    data_df  
    .select(upper('varsta'), upper('zona'), upper('extra'))  
)
```



```
from pyspark.sql.functions import col, lit, concat  
  
final_data_df = (  
    data_df  
    .select(concat(col('vechime'), ' ani' ) )  
)
```



# Column - Operatori

Pentru a construi formule mai complexe de calcul, obiectul Column oferă posibilitatea de a folosi operatorii comuni din Python pentru operații aritmetice și booleene și de comparare.

## ➤ Operatori elementari

<b>+</b>	Adunare	<b>-</b>	Scădere	<b>*</b>	Înmulțire	<b>/</b>	Împărțire	<b>~</b>	Negare booleană
<b>**</b>	Ridicare la putere	<b>%</b>	Restul împărțirii	<b>//</b>	Împărțire parte întreagă				

## ➤ Operatori de comparare

<b>&lt;</b>	<b>&gt;</b>	<b>&lt;=</b>	<b>&gt;=</b>	<b>==</b>	<b>!=</b>
-------------	-------------	--------------	--------------	-----------	-----------

❖ Toți operatorii permit și folosirea directă a valorilor în operații fără a fi nevoie de funcția `lit`

```
print((f.col('varsta') * f.lit(2) + f.col('vechime') - f.lit(3)) / f.lit(2) < 40)
```

```
Column<'((((varsta * 2) + vechime) - 3) / 2) < 40)'>
```

# Column - Funcții

Pentru a efectua operații pe caractere sau liste și alte structuri complexe, PySpark oferă funcții gata implementate.

- Funcțiile pot accepta un număr variabil de parametri, de la zero la doi sau mai mulți, în funcție de specificațiile ei.

contains	⇒	<code>print(f.contains(f.trim(f.col('nume')), f.lit('a')))</code>
trim		<code>Column&lt;'contains(trim(nume), a) '&gt;</code>

- Unele funcții acceptă doar parametrii de tip Column, altele necesită parametrii cu valori constante.

current_date	⇒	<code>print(f.date_trunc('DAY', f.current_date()) == '2024-01-01')</code>
date_trunc		<code>Column&lt;'(date_trunc(DAY, current_date()) = 2024-01-01) '&gt;</code>
array_remove	⇒	<code>print(f.size(f.array_remove(f.col('extra'), 'EV')) &gt; 0)</code>
size		<code>Column&lt;'array_except(extra, array(EV)) '&gt;</code>

Funcțiile și detaliile lor găsiți la <https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/functions.html>

# Column - Expresii SQL

Funcțiile oferite de Spark SQL sunt similare cu cele din bazele de date SQL. De asemenea, se pot parsea direct expresii în stil SQL care folosesc doar funcții disponibile în Spark SQL într-o expresie de tip Column.

- Metoda de parsare a unei expresii tip SQL dintr-un șir de caractere de Python.

```
expr = f.expr('concat("in varsta de ", varsta, " ani")')
```

- ❖ Funcția va returna un obiect de tip Column pentru expresia de calcul parsată din șirul de caractere:

```
print(expr)
```

```
Column<'concat(in varsta de , varsta,  ani) '>
```

- ✓ În acest exemplu, Spark expresia va concatena șirul "in vasta de " cu valoarea coloanei varsta și șirul " ani".

# Transformarea Datelor - With Column

Data Frame-urile oferă și metode pentru transformarea datelor, pe lângă cele de proiectare, scriere, colectare și afișare în același stil de apelare prin înlănțuire. Aceste metode se folosesc de expresiile de calcul.

➤ Operația de Transformare – Adăugarea sau Înlocuirea unui coloane

```
new_data_df = data_df.withColumn('text', f.expr('concat("in varsta de ", varsta, " ani")'))
```

```
new_data_df = data_df.withColumn('text', f.concat(f.lit('in varsta de '), f.col('varsta'), f.lit(' ani')))
```

❖ Un nou obiect de tip Data Frame este returnat care are adăugată sau actualizată coloana text cu rezultatul evaluării expresiei. Expresia se analizează și tipul coloanei este dedus automat.

```
new_data_df.show()
```

nume	varsta	ocupatie	vechime	inactiv	zona	extra	text
Vali	23	Programator	4	NULL	A	[3D Printer, XBOX]	in varsta de 23 ani
Vlad	34	Instalator	11	NULL	B	[EV]	in varsta de 34 ani
Bea	29	Reporter	7	true	B	NULL	in varsta de 29 ani



# Column - Tipuri de Date

Spark SQL nu poate determina tipul de date al obiectelor Column fără să cunoască tipurile valorilor implicate în expresie. Dar când expresia este integrată într-un Data Frame, ea este analizată și tipul de date este dedus.

- Obiectul Column dispune adițional de o metodă pentru a converti expresiile într-un tip de date specificat pentru cazul în care dorim să convertim rezultatul expresiei într-un alt tip de date.

```
expr = (f.col('varsta') + f.lit(1)).cast('string')
```

- ❖ Similar cu metodele din Data Frame, un nou obiect de tip Column este returnat cu o nouă expresie care convertește rezultatul expresiei anterioare la tipul de date specificat.

```
print(expr)
```

```
Column<'CAST((varsta + 1) AS STRING) '>
```

- ✓ În acest exemplu, Spark expresia va converti valoarea coloanei varsta + 1 într-un șir de caractere.

# Column - Nume

Fiecare obiect de tip Column vine automat cu un nume atribuit, care, prin implicit, este identic cu textul expresiei sale. Câteva funcții și metode din Spark utilizează acest nume, dar sunt relativ puține.

- Obiectul Column dispune adițional și de o metodă pentru a seta numele obiectului Column explicit.

```
expr = f.expr('concat("in varsta de ", varsta, " ani").alias('text_varsta')
```

- ❖ Similar cu metodele din Data Frame, un nou obiect de tip Column este returnat cu o nouă expresie care setează un nume expresiei precedente, similar cu metoda AS din SQL.

```
print(expr)
```

```
Column<'concat(in varsta de , varsta, ani) AS text_varsta'>
```

- ❑ Sunt relativ puține metode care se folosesc de numele expresiei. Una dintre ele este metoda `struct` care se folosește de numele expresiei în crearea structurii. Metoda `withColumn` ignoră complet numele.
- ❑ Numele expresiei nu se păstrează dacă o folosim pentru a crea alte expresii.

# Column - Condiții

Similar cu Data Frame-urile, metodele obiectului de tip Column returnează o coloană nouă. Așadar aceste metode pot fi înlănțuite. Metoda de scriere a condițiilor IF-ELSE se folosește de acest lucru.

## ➤ Condiții în lanț de IF-ELSE

```
print( f.when(f.col('varsta') < 25, f.lit('I')).when(f.col('varsta') < 32, f.lit('II')).otherwise(f.lit('III')) )
```

```
Column<'CASE WHEN (varsta < 25) THEN I WHEN (varsta < 32) THEN II ELSE III END'>
```

- ❖ Funcțiile `when` și `otherwise` ale tipului Column vor genera o eroare dacă sunt apelate fără a apela mai întâi funcția de calcul `when` din modulul de funcții.
- ❖ Dacă `otherwise` este omis, atunci expresia va rezulta în valoarea „NULL” dacă condițiile de `when` nu sunt îndeplinite.
- ❖ Evident, la finalul lanțului putem adăuga și funcțiile `cast` și `alias` dacă este necesar.

# Transformarea Datelor - Select

Similar cu interogările din bazele de date SQL, se pot efectua și transformări de date în funcția de select.

## ➤ Operația de Selecție – Selecție a coloanelor

```
new_data_df = data_df.select(
    'nume', 'extra', f.col('varsta') - f.col('vechime'),
    f.concat(f.lit('in varsta de '), f.col('varsta'), f.lit(' ani')).alias('text')
)
```

- ❖ Un nou obiect Data Frame este returnat care include doar coloanele specificate. Numele expresiilor este utilizat ca denumire pentru coloanele respective. Expresiile se analizează și tipul lor este dedus

```
new_data_df.show()
```

```
+-----+-----+-----+-----+
|nume|          extra|(varsta - vechime)|          text|
+-----+-----+-----+-----+
|Vali|[3D Printer, XBOX]|          19|in varsta de 23 ani|
|Vlad|          [EV]|          23|in varsta de 34 ani|
|Bea|          NULL|          22|in varsta de 29 ani|
+-----+-----+-----+-----+
```

# Un program simplu PySpark cu transformări de date

```
from pyspark.sql import SparkSession, functions as f

spark = SparkSession.builder.getOrCreate()

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

enriched_data_df = (
    data_df
    .withColumn('inactiv', f.coalesce(f.col('inactiv'), f.lit(False)))
    .withColumn('varsta_contractare', f.col('varsta') - f.col('vechime'))
)

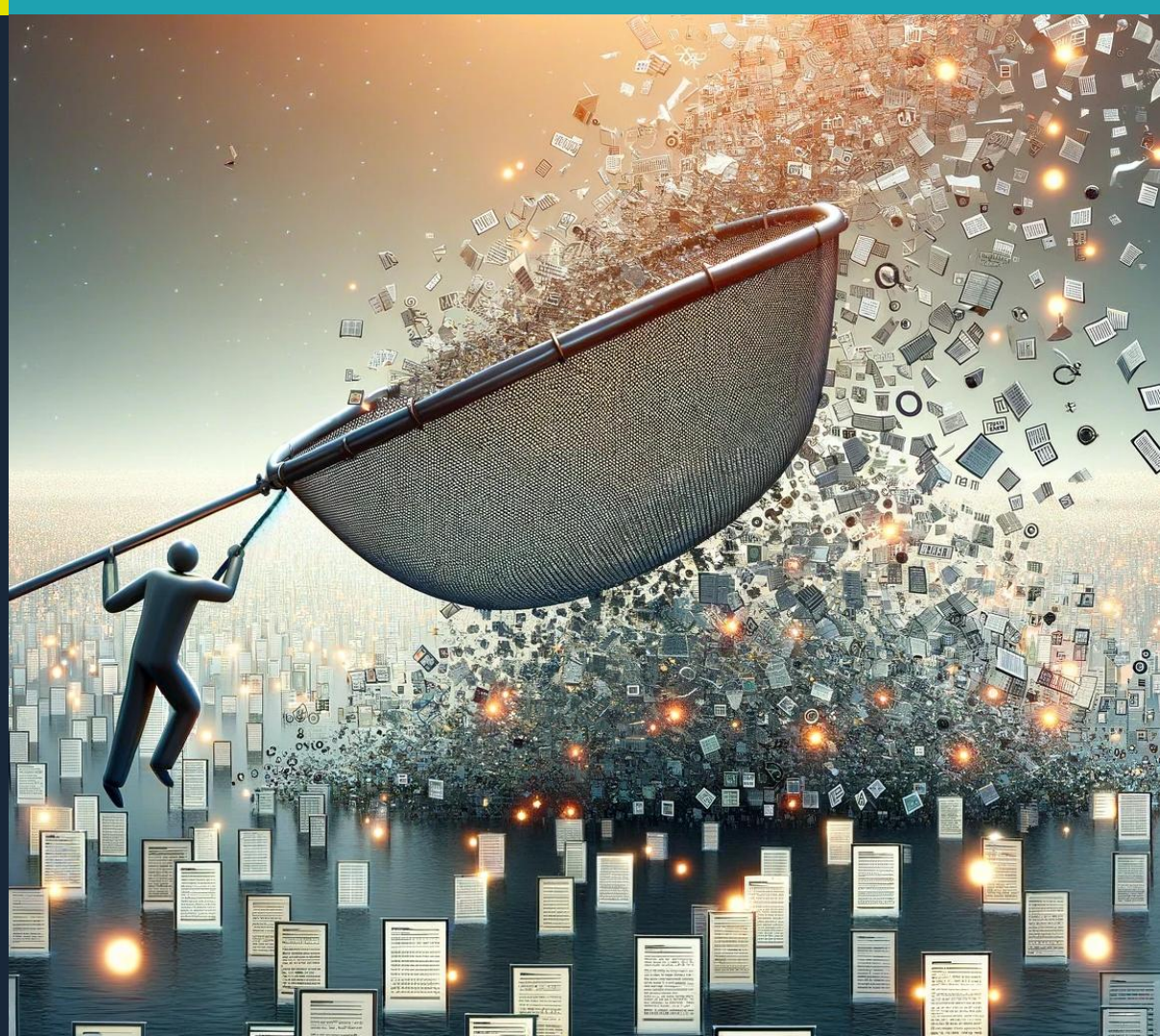
enriched_data_df.write.format('json').save('/path/to/save/folder')
```

# Metode de filtrare și ordonare

What's the catch?



Expert  
Services



# Filtrarea Datelor – Filter

Data Frame-urile oferă și metode pentru filtrarea datelor, pe lângă cele de transformare, scriere, colectare și afișare, similare cu interogările din bazele de date SQL, în același stil de apelare prin înlănțuire.

➤ Operația de Filtrare – Păstrarea datelor pe baza unei condiții

```
new_data_df = data_df.filter('varsta < 30')
```

```
new_data_df = data_df.filter(f.col('varsta') < 30)
```

```
new_data_df = data_df.where('varsta < 30')
```

```
new_data_df = data_df.where(f.col('varsta') < 30)
```

❖ Un nou obiect Data Frame este returnat care include doar datele ce îndeplinesc condiția specificată, valoare expresiei să fie diferită de False sau NULL. Metoda acceptă și expresii direct în format text.

```
new_data_df.show()
```

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

# Ordonarea Datelor – Sort

Implicit, datele sunt sortate după ordinea în care apar ele în fișiere sau din obiectele de Python furnizate. Spark permite schimbarea acestei ordine și oferă funcții speciale pentru setarea ei.

- Operația de Sortare – Ordonarea datelor după unul sau mai multe criterii de ordonare

```
new_data_df = data_df.sort('nume', f.desc(f.col('varsta')))
```

- ❖ Este returnat un nou obiect Data Frame ordonat după nume crescător, apoi vârstă descrescător la egalitate. Pot fi specificare oricâte criterii de sortare. În modulul de funcții se găsesc mai multe reguli de ordonare.

```
new_data_df.show()
```

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



# Limitarea Datelor – Limit

Pentru depanare, când se dorește testarea pe puține date, Spark prezintă și o metodă de a limita datele. Aceasta permite selecția rapidă a unui număr fix de date, fiind mai eficientă decât filtrarea, care procesează întregul set.

➤ Operația de Limitare – Limitarea numărului de rânduri

```
limited_data_df = data_df.limit(2)
```

❖ Un nou obiect Data Frame este returnat care include doar numărul de rânduri specificat, în ordinea implicită sau setată anterior cu `sort`. Adesea metoda este folosită pentru depanare și analiză sau înainte de colectare.

```
new_data_df = data_df.sort(f.desc('vechime')).limit(2)
```

```
new_data_df.show()
```

nume	varsta	ocupatie	vechime	inactiv	zona	extra
Vlad	34	Instalator	11	NULL	B	[EV]
Bea	29	Reporter	7	true	B	NULL

# Un program PySpark de analiză

```
from pyspark.sql import SparkSession, functions as f

spark = SparkSession.builder.getOrCreate()

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

enriched_data_df = (
    data_df
    .withColumn('inactiv', f.coalesce(f.col('inactiv'), f.lit(False)))
    .withColumn('varsta_contractare', f.col('varsta') - f.col('vechime'))
    .filter('inactiv is False')
    .sort(f.desc('vechime'))
    .limit(10)
)

enriched_data_df.write.format('json').save('/path/to/save/folder')
```

# Reutilizarea Operațiilor

Work smart, not hard



Expert  
Services



# Reutilizarea Operațiilor - Lanțuri comune de comenzi

De multe ori se întâmplă ca un lanț de comenzi, e.g. proiecții, transformări sau filtrări, să trebuiască refolosit.

```
data_df = initial_data_df.withColumn('varsta_contractare', f.col('varsta') - f.col('vechime'))
```



Citirea Datelor din `/path/to/course/data/folder`  
Adăugarea Coloanei `varsta_contractare = varsta - vechime`

`data_df`



```
data_df1 = data_df.filter('varsta_contractare < 22')
```



```
data_df2 = data_df.filter('varsta_contractare >= 22')
```



Citirea Datelor din `/path/to/course/data/folder`  
Adăugarea Coloanei `varsta_contractare = varsta - vechime`  
Păstrarea Datelor îndeplinesc condiția `varsta_contractare < 22`

`data_df1`



Citirea Datelor din `/path/to/course/data/folder`  
Adăugarea Coloanei `varsta_contractare = varsta - vechime`  
Păstrarea Datelor îndeplinesc condiția `varsta_contractare >= 22`

`data_df2`

La fiecare rulare, toate operațiile vor fi re-executate, ceea ce este inefficient pentru transformări complexe. Situația apare și dacă re-scriem același Data Frame, precum și atunci când Data Frame-urile au un istoric de comenzi comun.

# Reutilizarea Operațiilor – Cache

Pentru a reutiliza datele procesate de un lanț de comenzi, Spark oferă metode de a salva datele temporar pe executori, fără a fi nevoie de a întrerupe lanțul prin salvarea lor manuală de către dezvoltator.

- Operația de Caching – Stocarea datelor în memorie și surplusul pe disk până la închiderea sesiunii de Spark

```
cached_data_df = data_df.cache()
```

- ❖ Un nou obiect Data Frame este returnat cu o operație de salvare a datelor adăugată în lanțul de comenzi.
- ✓ La prima execuție a comenzilor unui Data Frame care continuă lanțul din `cached_data_df`, datele se salvează în memoria executorilor și surplusul pe disk. La o altă rulare, fie a aceluiași Data Frame, fie a unui alt Data Frame derivat, datele vor fi refolosite și operațiile de dinainte de `cache` nu vor mai fi executate.

```
new_data_df.explain('extended')
```

```
...  
== Optimized Logical Plan ==  
InMemoryRelation [inactiv#140, zona#146, extra#141, nume#142, ocupatie#143, varsta#144L, vechime#145L],  
StorageLevel(disk, memory, serialized, 1 replicas)  
...
```

# Reutilizarea Operațiilor – Persist

Spark pune la dispoziție și o metodă mai avansată de a salva datele temporar pe executori prin care se poate alege și modalitatea de salvare a datelor temporar.

- Operația de Persistență – Stocarea datelor după specificația furnizată până la închiderea sesiunii de Spark

```
from pyspark.storagelevel import StorageLevel
cached_data_df = data_df.persist(StorageLevel.MEMORY_ONLY)
```

- ❖ Un nou obiect Data Frame este returnat cu o operație de salvare a datelor, după opțiunile specificate, adăugată în lanțul de comenzi. Metoda se comportă identic cu metoda `cache`, însă acest caz specific, DOAR IN MEMORIE
- ✓ Metoda de `cache` utilizează această metodă în spate, specificând opțiunea de memorie și disk.

```
new_data_df.explain('extended')
```

```
...
== Optimized Logical Plan ==
InMemoryRelation [inactiv#140, zona#146, extra#141, nume#142, ocupatie#143, varsta#144L, vechime#145L],
StorageLevel(memory, 1 replicas)
...
```

# Reutilizarea Operațiilor - Exemplu

Metodele `cache` și `persist` sunt frecvent utilizate când este nevoie de divizarea setului de date, existând transformări standard ce trebuie realizate înainte de această împărțire.

```
cached_data_df = initial_data_df.withColumn('varsta_contractare', f.col('varsta') - f.col('vechime')).cache()
```

```
young_df = cached_data_df.filter('varsta_contractare < 22')  
young_df.write.format('parquet').save('/path/to/save/folder/1')
```

```
old_df = cached_data_df.filter('varsta_contractare >= 22')  
old_df.write.format('parquet').save('/path/to/save/folder/2')
```



Citirea Datelor din `/path/to/course/data/folder` young\_df  
Adăugarea Coloanei `varsta_contractare = varsta - vechime`  
Salvarea / Citirea datelor de pe executori  
Filtrarea datelor după condiția `varsta_contractare < 22`



Citirea Datelor din `/path/to/course/data/folder` old\_df  
Adăugarea Coloanei `varsta_contractare = varsta - vechime`  
Salvarea / Citirea datelor de pe executori  
Filtrarea datelor după condiția `varsta_contractare >= 22`

- ❖ La scrierea `young_df` se va efectua adăugarea coloanei și salva pe executori, operațiile continuând normal, iar la scrierea `old_df` datele din cache vor fi citite operațiile continuând din acel punct normal.



# Expert Service

Hands On Advanced Analytics with Apache Spark  
Practice