



# BIG DATA

CURS 3

# Plan curs

- **Map Reduce, Hadoop, Apache (recapitulare și completări)**
- **Apache Spark SQL**



# Review cursuri precedente

- Big Data (<https://www.gartner.com/en/information-technology/glossary/big-data>):

## Big Data

**Big data** is high-volume, high-velocity and/or high-variety information assets that demand cost-effective, innovative forms of information processing that enable enhanced insight, decision making, and process automation.

- Cei 3 V (pot ajunge până la 10 V):
  - Volum
  - Velocitate
  - Varietate
- Ce presupun „formele inovative de procesare a informației” din definiția Big Data?

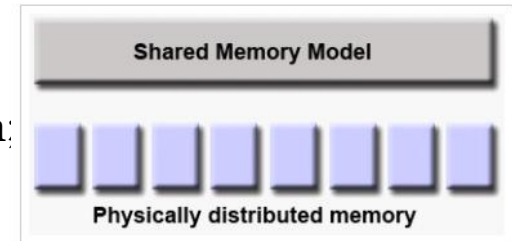
# Review cursuri precedente

- Procesare masivă => Mai multe sisteme de calcul
- Grace Hopper: *We shouldn't try for bigger computers, but for more systems of computers*
- Paralelismul este obligatoriu!

# Review cursuri precedente

Modele de programare paralelă:

- Transmisere de mesaje (*message passing*):
  - task-uri independente pe date locale;
  - task-urile interacționează prin schimb de mesaje
- Memorie partajată (*shared memory*):
  - task-urile partajează un spațiu de adrese de memorie comun;
  - task-urile interacționează citind/scriind din/în acest spațiu
- Paralelizarea datelor (*data parallelization*)
  - Task-urile execută operații independente pe partiții de date
  - Model foarte potrivit pentru probleme „extrem de paralele” (*embarrassingly parallel*)



# Review cursuri precedente

## Infrastructuri tradiționale

Stocare de  
date



Bază de date

Cluster de calcul



...



Noduri de calcul

Rezultat



# Review cursuri precedente

## Message Passing / Shared Memory

Stocare de  
date



Cluster de calcul



...



Fiecare nod poate încărca  
datele complete

- Nu scalează

Rezultat



# Review cursuri precedente

## Paralelizarea datelor

Stocare de  
date



Cluster de calcul



...



Fiecare nod încarcă doar o partiție

- Scalare mai bună
- Încă necesită transferul prin rețea al tuturor datelor

Rezultat



# Review cursuri precedente

## **Soluție: localizarea datelor**

- Dacă deplasarea datelor este o problemă, ar trebui să nu mai deplasăm date!
- Soluția nu este suportată de infrastructurile / cluster-ele tradiționale
  - Sunt necesare cluster-e care să partajeze CPU, nu stocare
  - Stocarea trebuie realizată pentru operații I/O, nu pentru calcule

# Review cursuri precedente

## Conceptul cheie al tehnologiilor Big Data

Paralelizare



Localizarea datelor



Cluster de calcul cu stocare distribuită



Exemple:



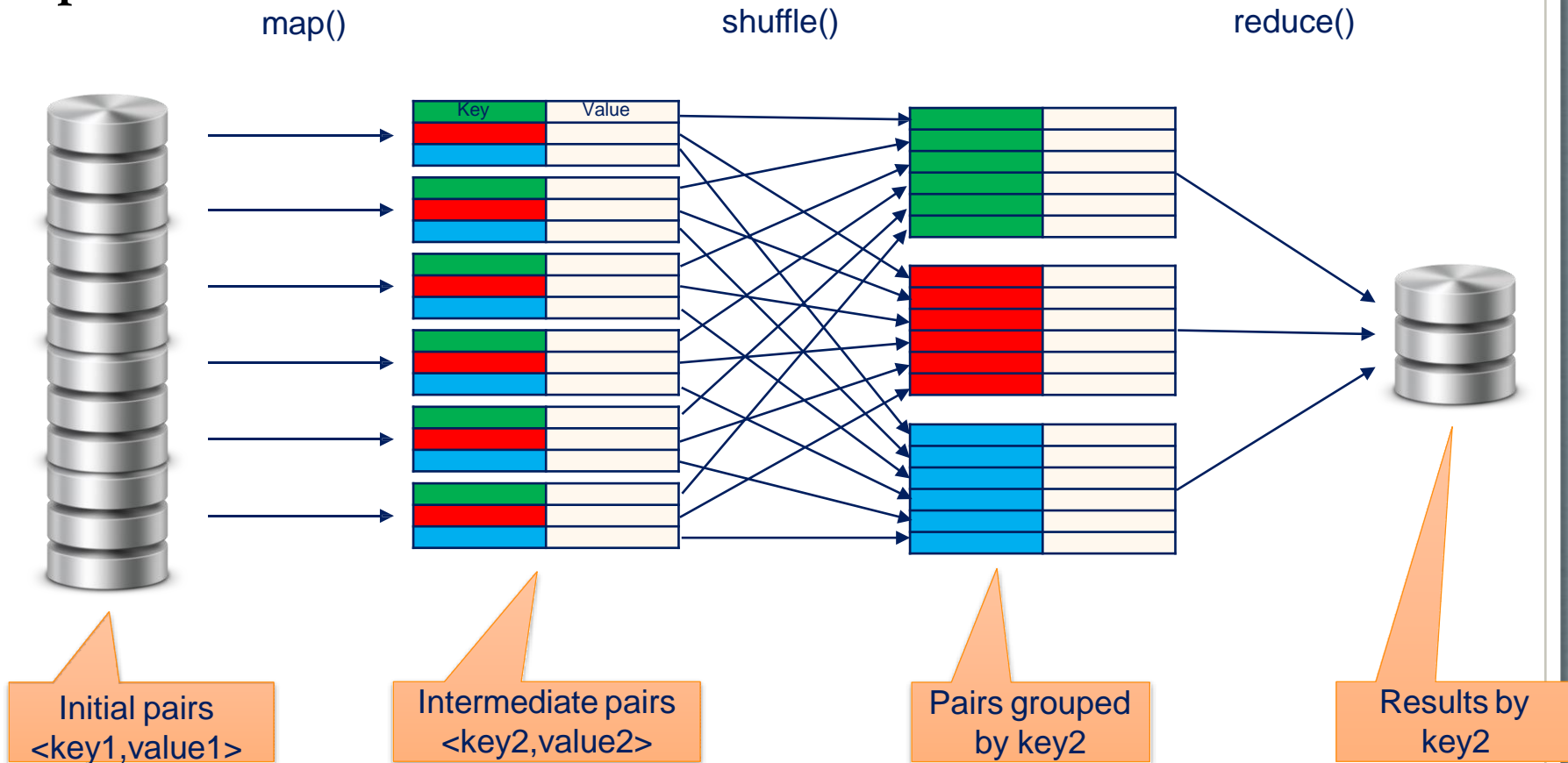
# Review cursuri precedente

## Map Reduce

- Model de programare pentru paralelizarea datelor
  - Publicat de Google în 2004
- Furnizează funcțiile `map()` și `reduce()` pentru procesarea datelor
  - Se bazează pe transformări cheie-valoare
- Furnizează funcția `shuffle()` pentru aranjarea elementelor intermediare
- Distribuirea are loc conform paradigmei master/worker
  - Suportă disponibilitatea înaltă și recuperarea
  - A fost discutat anterior (Hadoop)

# Review cursuri precedente

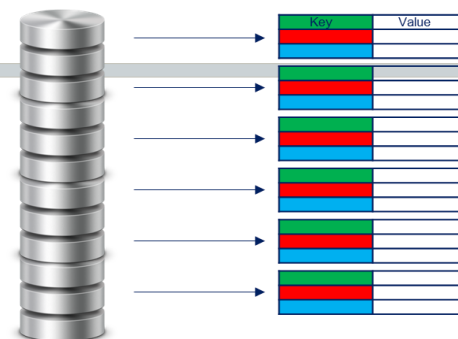
## Map Reduce



# Review cursuri precedente

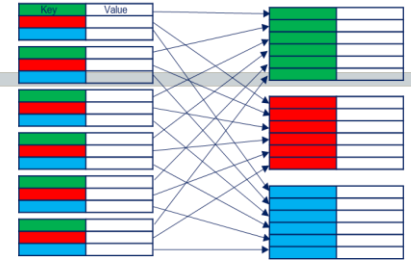
## Funcția map()

- Concept din programare funcțională
- Aplică o funcție **separat** pe fiecare item din intrare
  - `map(funcție, <key1, value1>) -> list(<key2, value2>)`
  - de obicei, funcțiile sunt definite de utilizator
- Cheile de intrare și cele de ieșire pot fi diferite
  - Tipurile de date pot fi, de asemenea, diferite
- Ieșirea este o listă => o mapare poate avea ieșiri multiple
  - Toate elementele listei trebuie să aibă același tip de date



În implementarea MapReduce inițială, toate cheile și valorile erau șiruri de caractere. Utilizatorii trebuia să realizeze conversiile necesare în cadrul funcțiilor map/reduce.

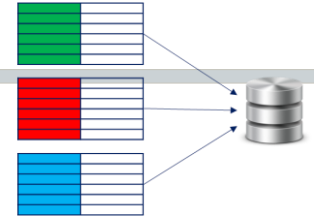
# Review cursuri precedente



## Funcția `shuffle()`

- Organizează datele în funcție de chei
  - `shuffle(list(<key2, value2>) -> list(<key2, list(value2)>))`
- Frecvent, include sortarea după cheie pentru eficiență
- Funcția `shuffle()` nu așteaptă încheierea funcției `map()`
  - De îndată ce o pereche `<key2, value2>` este disponibilă, aceasta poate fi amestecată (*shuffled*)
  - Reduce timpii de așteptare
- Este furnizată de framework-ul MapReduce
  - Poate fi suprascrisă de utilizator pentru a efectua optimizări specifice *use case*-ului respectiv

# Review cursuri precedente



## Funcția reduce()

- Asemănătoare operației fold din programarea funcțională
- Agregă perechile  $\langle \text{key2}, \text{value2} \rangle$  care au aceeași cheie
  - Se obține o singură valoare pentru fiecare cheie
- Rezultă o listă de valori, câte una pentru fiecare cheie
  - `reduce(funcție, list( $\langle \text{key2}, \text{list}(\text{value2}) \rangle$ )) -> list(value2)`
  - de obicei, funcțiile sunt definite de utilizator

# Review cursuri precedente

## Paralelizarea cu ajutorul MapReduce

- Intrarea poate fi citită pe fragmente (*chunks*)
  - Paralelism pentru crearea perechilor cheie-valoare inițiale
- map() poate fi calculată independent pentru fiecare pereche cheie-valoare
- shuffle() poate începe să se execute imediat după ce a fost procesată prima pereche cheie-valoare
  - Limitează timpii de așteptare
- reduce() poate rula în paralel pentru diferite chei
  - Nu este necesar să aștepte încheierea execuției lui map()
  - Poate porni execuția atunci când sunt disponibile toate rezultatele corespunzătoare unei chei

# Review cursuri precedente



„Hello world” pentru MapReduce

## Numărarea cuvintelor cu ajutorul MapReduce

- Datele de intrare: 

How do you feel?  
Today I feel fine, today is a new day.
- Perechile inițiale  $\langle \text{key1}, \text{value1} \rangle$  sunt:
  - $\langle \text{line1}, \text{“how do you feel”} \rangle$
  - $\langle \text{line2}, \text{“today I feel fine today is a new day”} \rangle$
- Funcția `map()`: generează perechi  $\langle \text{word}, 1 \rangle$  pentru fiecare cuvânt din fiecare linie
  - $\langle \text{line1}, \text{“how do you feel”} \rangle$ 
    - $\langle \text{“how”}, 1 \rangle, \langle \text{“do”}, 1 \rangle, \langle \text{“you”}, 1 \rangle, \langle \text{“feel”}, 1 \rangle$
  - $\langle \text{line2}, \text{“today I feel fine today is a new day”} \rangle$ 
    - $\langle \text{“today”}, 1 \rangle, \langle \text{“I”}, 1 \rangle, \langle \text{“feel”}, 1 \rangle, \langle \text{“fine”}, 1 \rangle, \langle \text{“today”}, 1 \rangle, \langle \text{“is”}, 1 \rangle, \langle \text{“a”}, 1 \rangle, \langle \text{“new”}, 1 \rangle, \langle \text{“day”}, 1 \rangle$
- Funcția `reduce()`: cheilor le vor fi concatenate sumele valorilor:
  - $\langle \text{“how”}, \text{list}(1) \rangle \rightarrow \text{“how: 1”}$
  - $\langle \text{“today”}, \text{list}(1,1) \rangle \rightarrow \text{“today: 2”}$
  - ...

# Review cursuri precedente



## Apache Hadoop

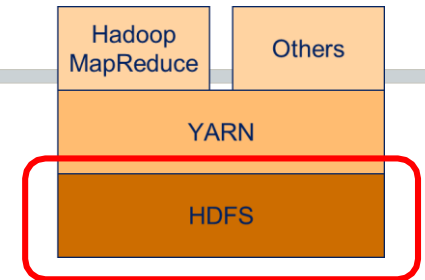
- Implementare open-source a lui MapReduce
  - Suportat de furnizorii principali de cloud
  - Utilizat de multe companii mari (Twitter, Facebook, Amazon)

Data Processing	Hadoop MapReduce	Others
Cluster resource management	YARN	
Distributed file system	HDFS	

# Review cursuri precedente

## Hadoop Distributed File System (HDFS)

- Componentă de bază a lui Hadoop
- Obiective ale HDFS:
  - Procesare crescută
  - Suport pentru fișiere și seturi de date mari
  - Mutarea procesării în loc de mutarea datelor (-> localizarea datelor)
  - Rezistență la defectele hardware
- Utilizează o arhitectură master/slave



# Review cursuri precedente

## Hadoop Distributed File System (HDFS)

Users



NameNode



- Punct de acces pentru clienți
- Expune operațiunile sistemului de fișiere
- Organizează crearea / ștergerea / replicarea blocurilor din DataNodes
- Poate exista un NameNode secundar pentru a evita un unic punct de cădere

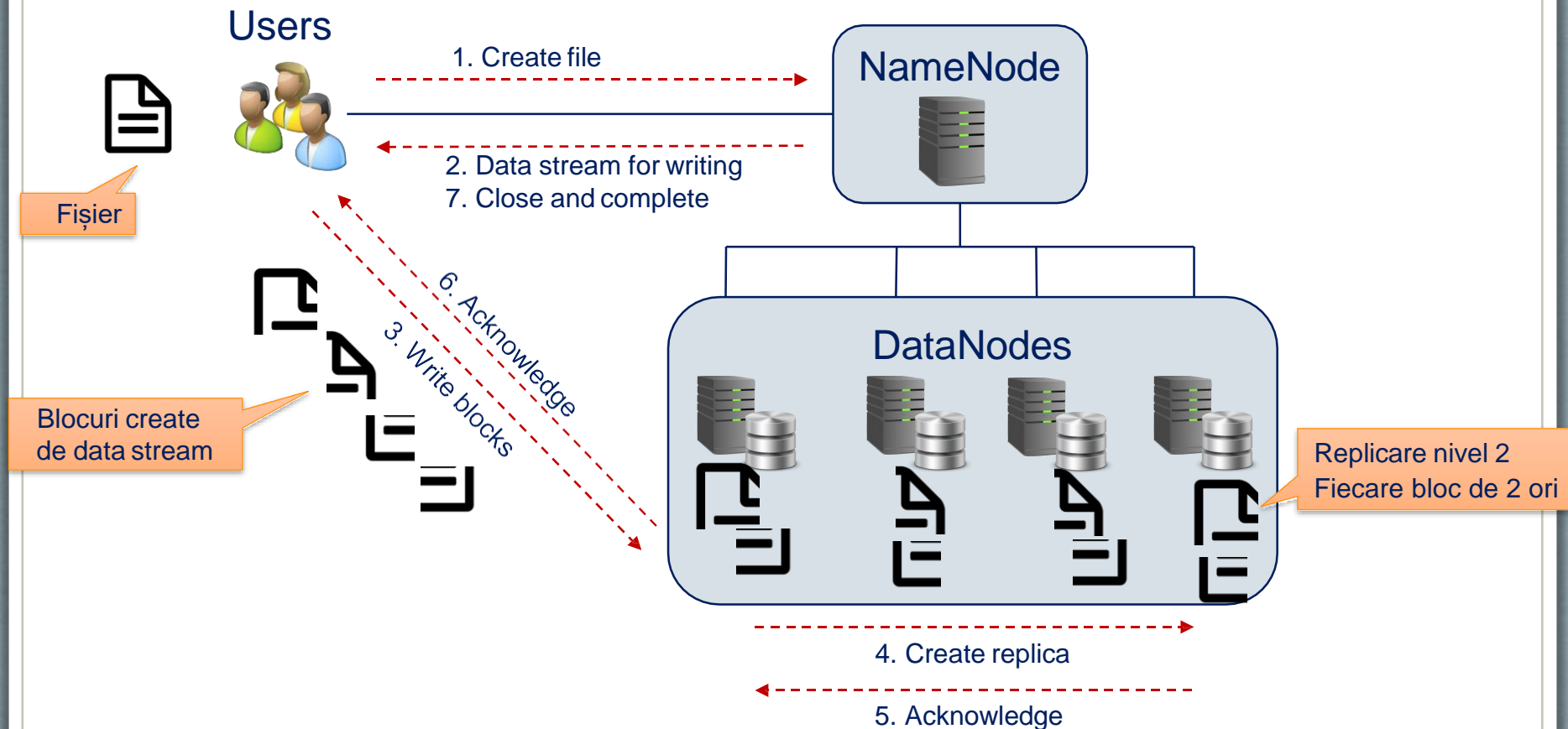
DataNodes



- Stochează blocurile de date
- Răspunde cererilor de citire / scriere
- Efectuează calcule pe blocuri

# Review cursuri precedente

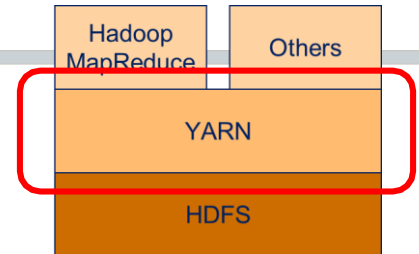
## Exemplu: scriere fișier



# Review cursuri precedente

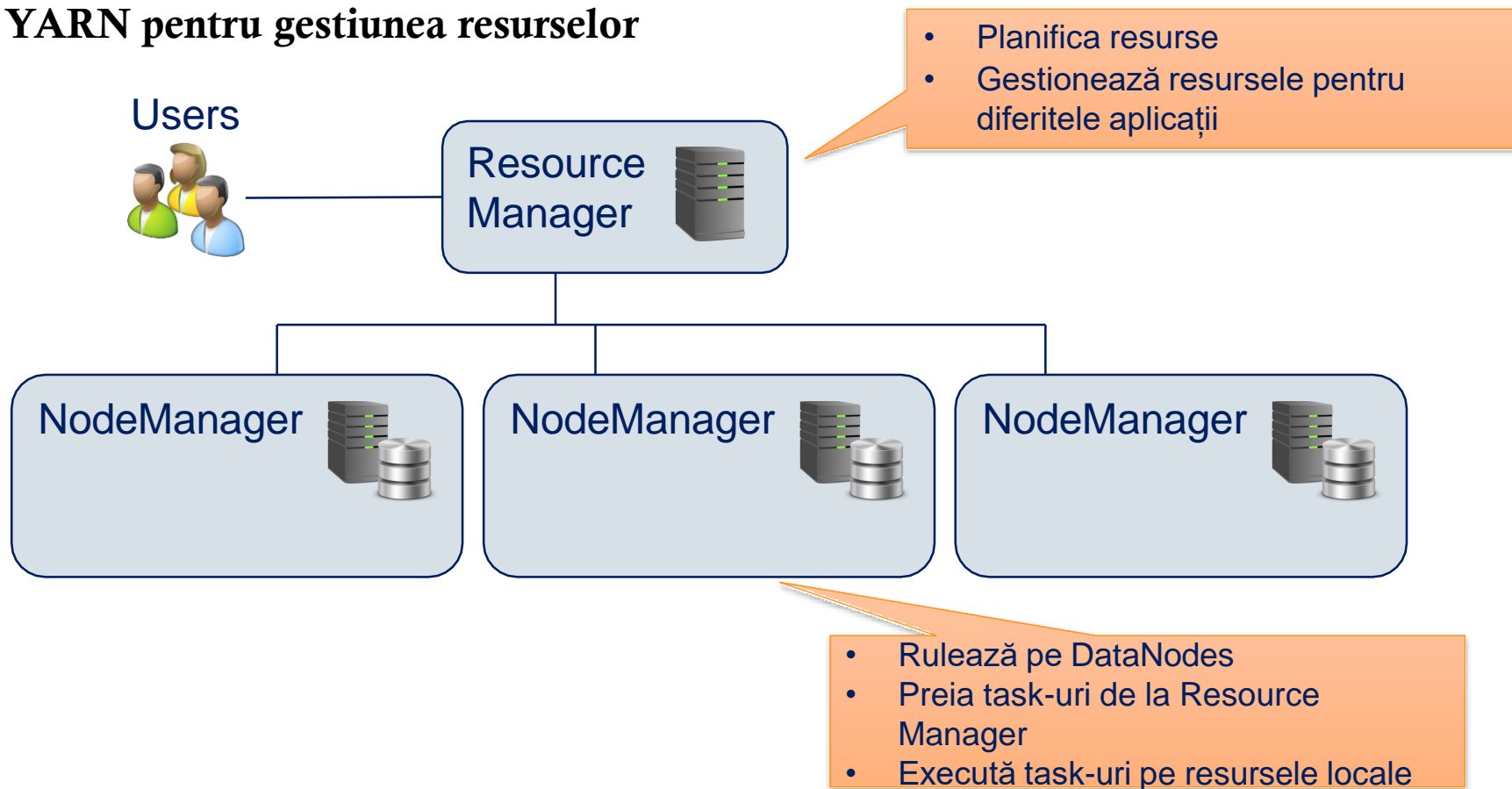
## Procesare cu Hadoop

- HDFS doar distribuie stocarea blocurilor
- Fiecare nod trebuie să aibă și rol de nod de calcul
  - Task-uri Map/Reduce/Shuffle
  - Preferabil și task-uri generale de calcul
- Fiecare resursă trebuie să fie utilizată de un singur task
  - Core-uri CPU
  - Memorie
    - Fără suprautilizare
- Resursele trebuie să fie folosite cât mai eficient posibil



# Review cursuri precedente

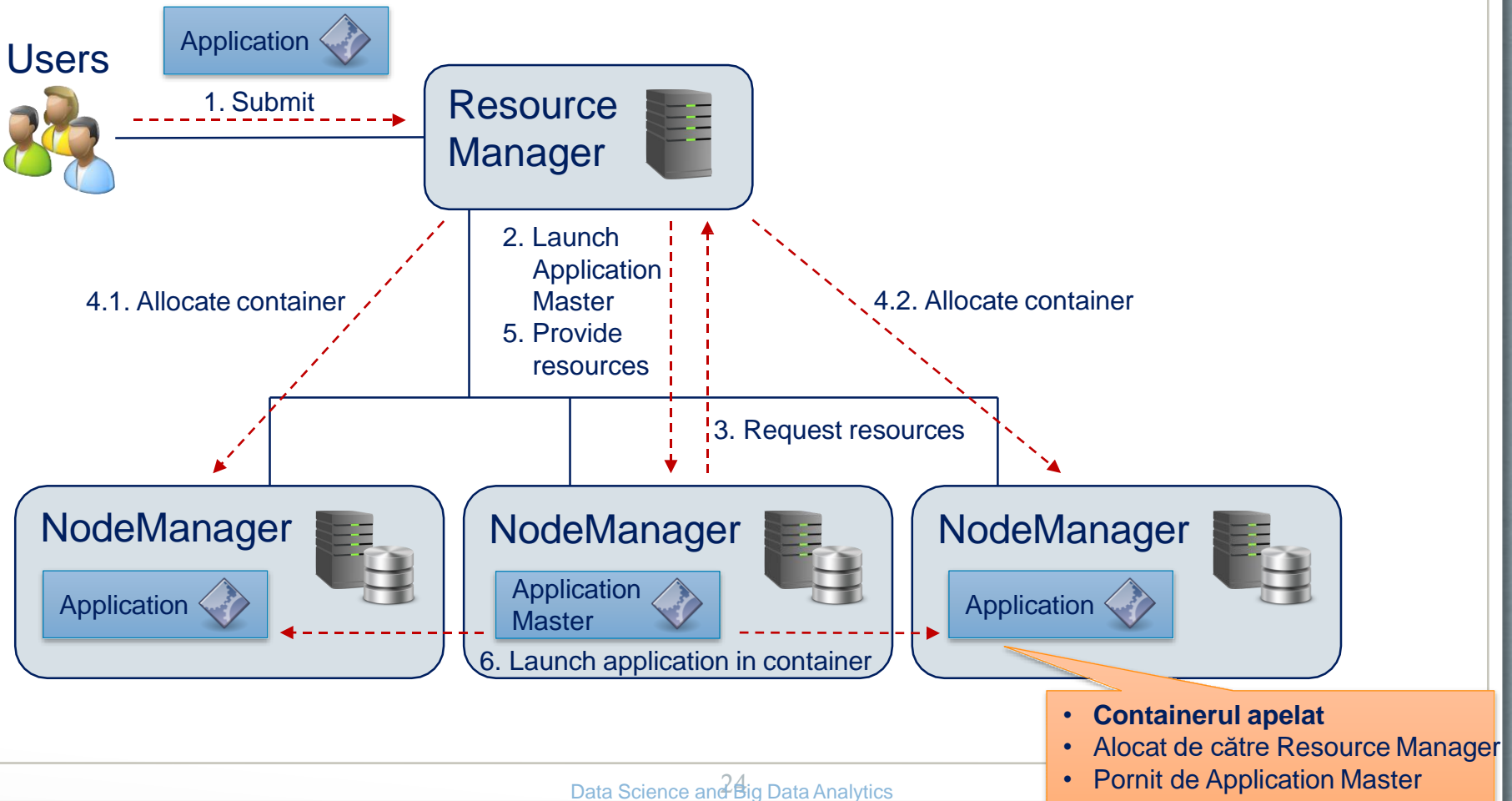
## YARN pentru gestiunea resurselor



YARN = Yet Another Resource Negotiator

# Review cursuri precedente

## Rularea aplicațiilor cu YARN



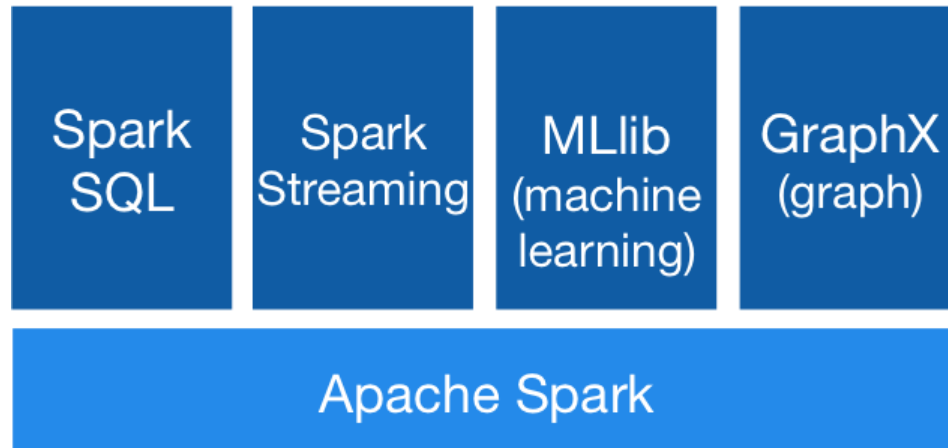
# Review cursuri precedente



## Apache Spark

- Motor pentru procesarea pe scară largă a datelor
- Proiectat pentru rezolvarea limitărilor din Hadoop pentru analiza datelor
- Suportă analiza in-memory
  - Suport bun pentru algoritmi iterativi
- Suportă combinații arbitrare de task-uri Map și Reduce
  - Utilizatorii nu trebuie să prevadă job-uri specifice și dependențele lor

# Spark Stack



- SparkSQL – pentru cereri de tip SQL și generare de *data frame*-uri
- Spark Streaming – pentru procesarea live a datelor de tip *streaming*
- MLlib – pentru algoritmi de Machine Learning
  - > 20 algoritmi pentru clustering, regresie și clasificare
- GraphX – pentru algoritmi pe grafuri

# Structuri de date utilizate în Spark

- Nu există un sistem de fişiere precum cel din Hadoop, ci 2 structuri de date *in-memory* importante
- *Resilient Distributed Dataset* (RDD)
  - Nivel de abstractizare pentru operaţiile asupra datelor
  - Partiţii imutabile de elemente
  - Toate elementele unui RDD pot fi procesate în paralel
  - Suportă operaţiile map, reduce, filtrare, funcţii definite de utilizator şi persistenţă
- *Data frame*
  - Un nivel mai ridicat de abstractizare, construit peste RDD
  - Similar dataframe-urilor din R / pandas
  - De obicei, generat utilizând API-ul SparkSQL

# Infrastructuri pentru execuția Apache Spark

- Spark permite crearea de clustere pentru calcul
  - Nu este modalitatea standard de utilizare a lui Spark
- Compatibil cu multe infrastructuri existente
- Computing
  - Hadoop/YARN și Apache Mezos
- Stocare
  - Hadoop/HDFS, Cassandra, Hbase, MongoDB etc.

# Programare cu Apache Spark

- Implementat nativ în Scala
  - Limbaj JVM cu concepte referitoare la inferență de tip și programare funcțională
- Furnizează API-uri pentru
  - Java
  - Python (PySpark)
  - R (SparkR)

# Numărarea cuvintelor în PySpark

```
text_file = sc.textFile("hdfs://data.txt")
counts = text_file.flatMap(lambda line: line.split(" ")) \
    .map(lambda word: (word, 1)) \
    .reduceByKey(lambda a, b: a + b)
counts.saveAsTextFile("hdfs://word_
count.txt")
```

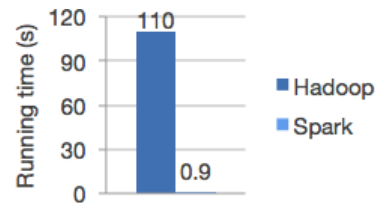
flatMap poate mapa intrarea în output-uri multiple

reduceByKey reduce setul de date prin fuziunea (reducerea) perechilor cheie-valoare care au aceeași cheie

Funcțiile lambda din Python: funcții anonime cu parametri a, b, ce returnează rezultatul expresiei specificate după „:” (a+b)

# Diferențe majore față de Hadoop

- In-memory
  - RDD-urile și Data frame-urile sunt prelucrate *in-memory*, dacă acest lucru este posibil
  - Îmbunătățirea performanțelor
    - Exemplu: Regresie logistică
- Nu furnizează un sistem de stocare distribuit propriu



# Ecosistemul Spark

- Diferite moduri de execuție
- Comunitate open-source considerabilă
- Multe tehnologii construite pe baza Spark
  - <https://spark-packages.org/>
- Dezvoltare rapidă, în desfășurare
  - Conceptele centrale sunt stabile

# Concluzii Review

- Procesarea Big Data necesită infrastructuri dedicate
  - Deplasarea datelor nu este fezabilă
  - Se deplasează procesarea (calculul) acolo unde sunt datele
- MapReduce este un model de programare pentru procesarea big data
- Apache Hadoop este un framework pentru Big data
  - HDFS – sistem de fișiere distribuit și fiabil
  - YARN – pentru managementul resurselor
  - Furnizează un framework MapReduce
- Apache Spark permite procesarea in-memory pentru big data
  - Compatibil cu diferite infrastructuri, inclusiv Hadoop
  - Furnizează API pentru machine learning



## **Apache Spark SQL**

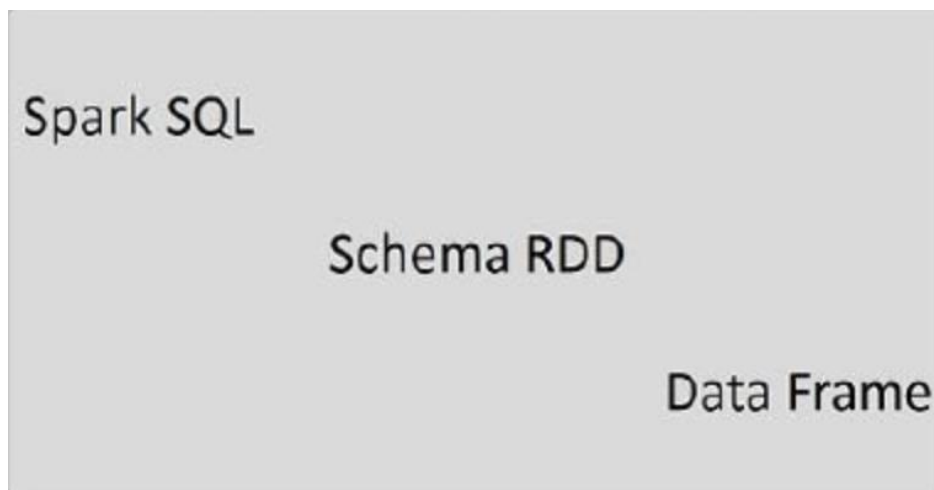
- Una dintre cele 4 componente majore ale lui Apache Spark
- Modulul Spark pentru lucrul cu date structurate

# Spark SQL

- Spark – printre tehnologiile open-source la care s-au adus cele mai multe contribuții
- Spark SQL – modulul Spark la care s-au adus cele mai multe contribuții

# Motivarea Spark SQL (1)

- Multe aplicații gestionează **date structurate**
  - Mai ales după curățarea datelor brute
- Utilizarea de structuri facilitează **optimizarea**

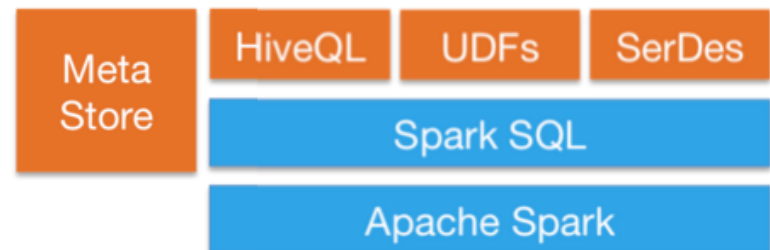


# Motivarea Spark SQL (2)

- **Compatibilitate cu Hive**

- Compatibilitate full cu datele, cererile și UDF-urile Hive existente

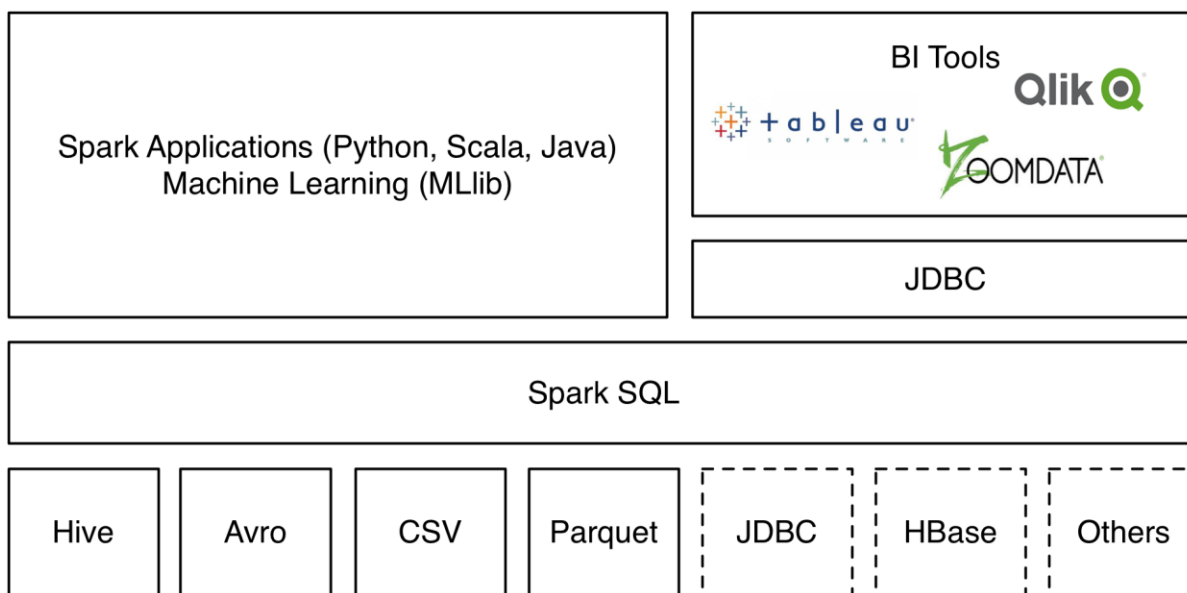
```
context = HiveContext(sc)
results = context.sql(
    "SELECT * FROM people")
names = results.map(lambda p: p.name)
```



- **Funcționalități SQL integrate**

# Motivarea Spark SQL (3)

- **Acces uniform la date**
  - Permite conexiunea la orice sursă de date: Hive, Avro, Parquet, JSON, JDBC etc.



# Evoluția lui Spark SQL (1)

- **MapReduce:**
  - Nivel jos, interfață de programare procedurală
  - Mult cod sursă, optimizare manuală
- PIG, Hive, Dremel etc:
  - Interfețe relaționale pentru Big Data
  - Cereri declarative; optimizări
  - Rulează pe MapReduce; Nu au suport pentru algoritmi complecși



# Evoluția lui Spark SQL (2)

- **Shark:**

- Utilizează motorul Hive pe Spark
- Funcționalitate limitată:
  - Interoghează doar datele externe stocate în catalogul Hive
  - Optimizorul Hive a fost proiectat pentru MapReduce
    - Dificil de extins pentru algoritmi Machine Learning



- **Spark SQL**



# Obiective

- Suportă procesare relațională în:
  - Programele Spark (RDD-uri native)
  - Surse de date externe
- Furnizează o performanță ridicată utilizând tehnicile existente în SGBD-uri
- Suportă noi surse de date, inclusiv date semistructurate și baze de date externe
- Permit extensii pentru algoritmi analitici avansați

# Componente centrale

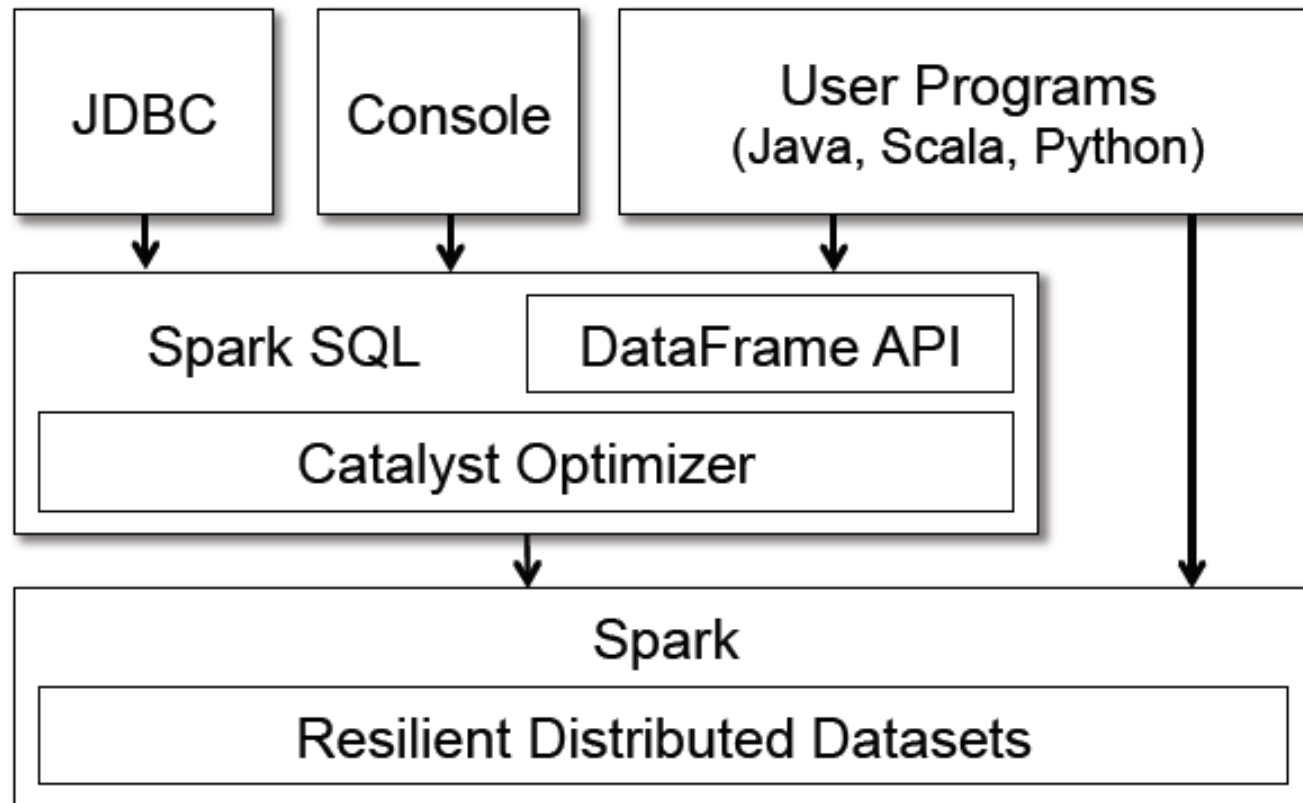
- Interfețe procedurale și relaționale
  - Pot fi combinate
  - RDD-uri structurate

⇒ DATAFRAME API

- Optimizare la nivelul SGBD
- Permite utilizatorilor să scrie reguli de optimizare

⇒ CATALYST OPTIMIZER

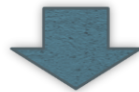
# Interfața de programare



# DataFrame API (1)

- DataFrame = RDD + Schema
  - Colecție distribuită de date **organizate pe coloane**
  - Similar unui tabel dintr-o bază de date relațională

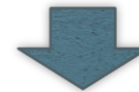
**Blackbox to Spark**



**RDD**

```
B6 01 F0 B0 00 00 00 00 00 |
00 2F 00 00 00 24 00 04 00 |
01 B2 4C 2B C7 00 05 01 B0 |
01 9C B7 01 B6 B0 00 00 00 |
00 01 00 2F 00 00 00 11 00 |
2A B4 01 A0 B0 00 00 00 00 |
01 00 2F 00 00 00 14 00 02 |
```

**Known structure to Spark**



**DataFrame**

dept	age	name
Bio	48	H Smith
CS	54	A Turing
Bio	43	B Jones
Chem	61	M Kennedy

Data grouped into  
named columns

# DataFrame API (2)

- Exemplu

```
ctx = new HiveContext()  
users = ctx.table("users")  
young = users.where(users("age") < 21)  
println(young.count())
```

Schema este obținută  
din Hive



DataFrame



# Workflow SQL + Procedural

- Se pot aplica comenzi SQL pe DataFrame

```
context = HiveContext(sc)
results = context.sql(
    "SELECT * FROM people")
names = results.map(lambda p: p.name)
```

- Se poate aplica un workflow procedural folosind construcții SQL

```
ctx = new HiveContext()
users = ctx.table("users")
young = users.where(users("age") < 21)
println(young.count())
```


# Modelul de date

- Utilizează modelul de date imbricat
- Suportă toate tipurile de date majore din SQL:
  - Primitive: boolean, integer, double, decimal, string, date, timestamp
  - Complexe: structs, arrays, maps, unions
- Suportă funcții definite de utilizator (UDF – User Defined Functions)

# Operații pe Dataframe-uri (1)

- Suportă toate operațiile majore relaționale:
  - Select, join, where, groupBy etc.
- **Exemplu:** calculul numărului de angajați femei din fiecare departament

Dataframes



```
employees  
  .join(dept, employees("deptId") === dept("id"))  
  .where(employees("gender") === "female")  
  .groupBy(dept("id"), dept("name"))  
  .agg(count("name"))
```

# Operații pe Dataframe-uri (2)

- Pot fi înregistrate ca tabele temporare și interogate cu ajutorul SQL
- **Exemplu:**

Construcții SQL procedurale

Înregistrarea output-ului ca TempTable

```
users.where(users("age") < 21)
      .registerTempTable("young")
ctx.sql("SELECT count(*), avg(age) FROM young")
```




SQL query

The diagram illustrates the process of registering a DataFrame as a temporary table and then querying it. Three blue arrows point from descriptive text to specific parts of the code: one from 'Construcții SQL procedurale' to the first line, one from 'Înregistrarea output-ului ca TempTable' to the second line, and one from 'SQL query' to the third line.

# Dataframe-uri vs Limbaje de interogare relaționale

- DataFrames = Relațional + Procedural
  - => SQL + Java, Scala sau Python
  - => SQL + instrucțiuni de ciclare, condiționale etc.
- Utilizatorii pot descompune codul sursă în funcții Scala, Java sau Python care transmit obiecte DataFrame între ele, construind astfel un plan logic al aplicației
- Beneficii legate de optimizare în cadrul acestui plan

# Interogarea seturilor de date native

- Construirea RDD-urilor => construirea DataFrame-urilor
- Scala  și Java :
  - Informația despre tip este extrasă din limbaj (sistemul său de typing)
- Python :
  - Inferență pe schemă, din cauza faptului că limbajul este unul cu tipuri dinamice

```
case class User(name: String, age: Int)

// Create an RDD of User objects
usersRDD = spark.parallelize(
  List(User("Alice", 22), User("Bob", 19)))

// View the RDD as a DataFrame
usersDF = usersRDD.toDF

views = ctx.table("pageviews")
usersDF.join(views, usersDF("name") === views("user"))
```

# Cache *in-memory*

- Materializează (reține în cache) în memorie datele accesate, utilizând stocarea pe bază de coloane
- Utilizează compresia pe coloane (Columnar)
  - Compresie a memoriei 10X
- Beneficii majore pentru algoritmi iterativi
  - Pot fi stocate mai multe date în memorie
  - Machine Learning și procesare de grafuri

# Funcții definite de utilizator

- Sistemele tradiționale de baze de date
  - Utilizează funcții definite de utilizator pentru suportul JSON, algoritmi ML
  - Se scriu în diferite medii de programare
  - Necesită încapsulare, înregistrare și import
- În Spark SQL
  - Se pot înregistra inline cu funcțiile Scala, Java sau Python
  - Pot fi folosite utilitățile BI prin interfața JDBC/ODBC

```
val model: LogisticRegressionModel = ...
```

```
ctx.udf.register("predict",  
  (x: Float, y: Float) => model.predict(Vector(x, y)))
```

```
ctx.sql("SELECT predict(age, weight) FROM users")
```

# *Catalyst Optimizer*

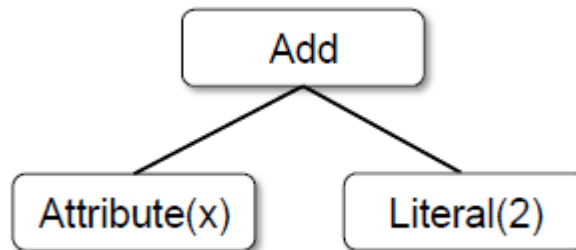
## **Ce este Catalyst Optimizer?**

- Un optimizor extensibil pentru Spark SQL
- Optimizor pe bază de reguli

## **Scop:**

- Facilitează adăugarea de caracteristici și tehnici de optimizare
- Se poate extinde prin adăugarea de noi reguli

# Arbori *Catalyst*

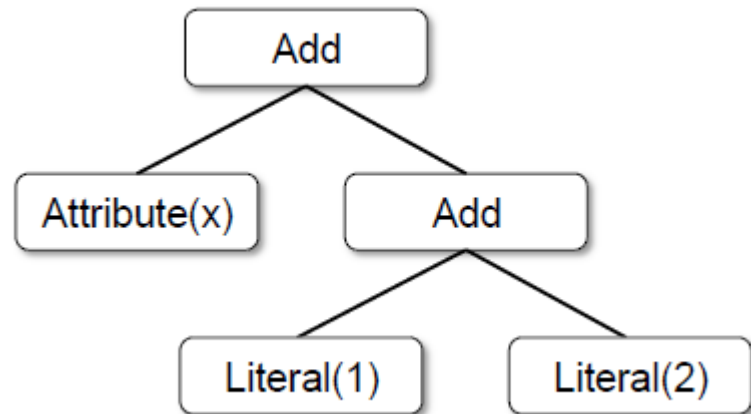


- Tipul de date principal în Catalyst este arborele
- Alcătuit din noduri obiect
- Fiecare nod are asociat un tip de nod și zero sau mai mulți copii
- Noile tipuri de noduri pot fi definite ca subclase ale clasei `TreeNode`
- Obiectele nod sunt imutabile
- Arborii pot fi prelucrați utilizând transformări funcționale (reguli).

# Exemplu de arbore *Catalyst*

Arbore pentru  
expresia:

$x + (1 + 2)$



Reprezentare Scala:

`Add(Attribute(x), Add(Literal(1), Literal(2)))`

# Transformarea arborilor folosind reguli (1)

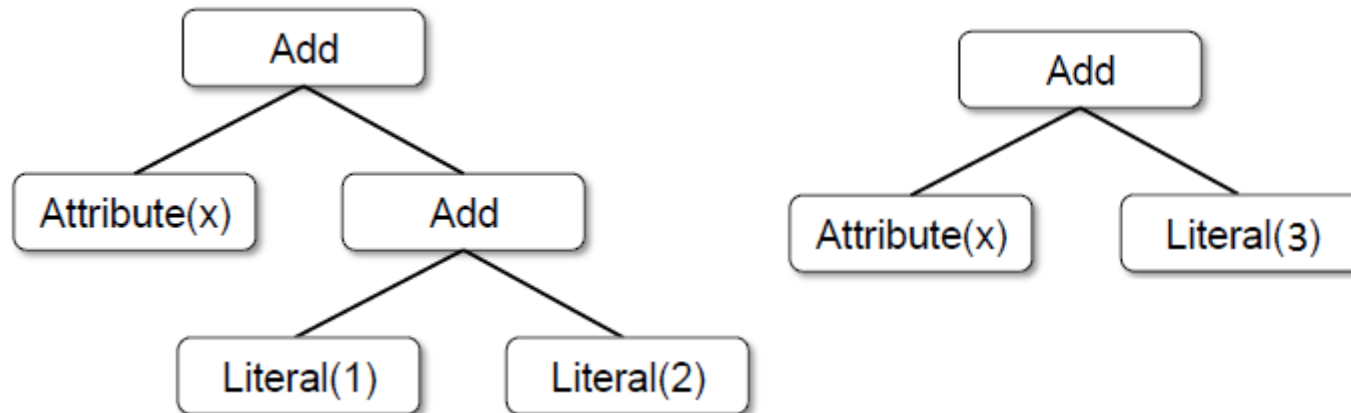
- Transformarea arborilor
  - Reguli care transformă un arbore în alt arbore echivalent
- Se folosesc funcții de patern matching pentru a găsi și înlocui subarbori cu o structură specifică
- Regulile de transformare din SGBD-uri sunt deja încorporate

# Transformarea arborilor folosind reguli (2)

- Presupunem că avem o regulă care tratează operația Add între constante

```
tree.transform{  
  case Add(Literal(c1), Literal(c2)) => Literal(c1 + c2)  
}
```

- Aplicând această regulă expresiei  $x + (1 + 2)$  vom obține un nou arbore  $x + 3$



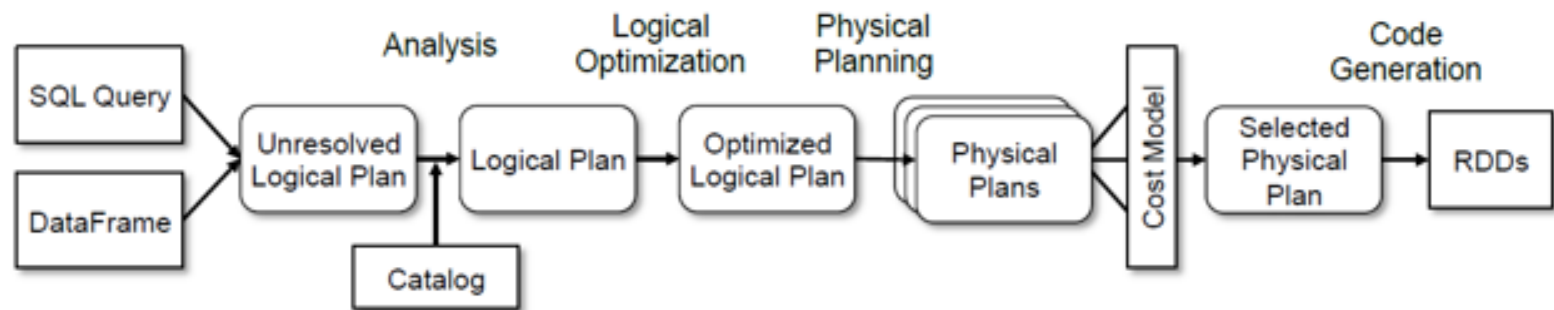
# Transformarea arborilor folosind reguli (3)

- Putem defini pattern-uri multiple în cadrul aceleiași transformări:

```
tree.transform{  
  case Add(Literal(c1), Literal(c2)) => Literal(c1 + c2)  
  case Add(left, Literal(0)) => left  
  case Add(literal(0), right) => right  
}
```

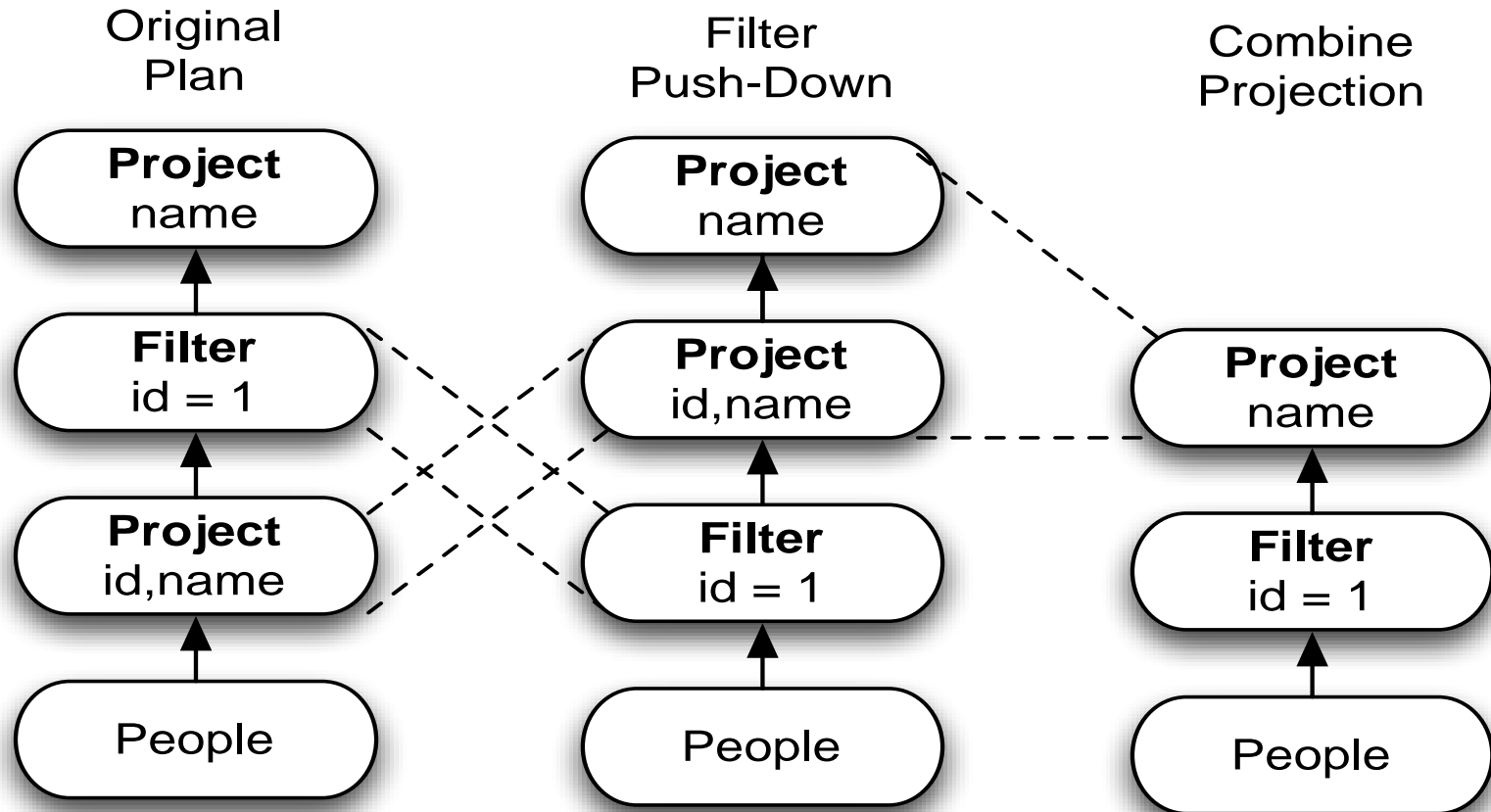
# Utilizarea Catalyst în Spark SQL

- Cadrul de lucru cu arbori Catalyst este utilizat în 4 etape:



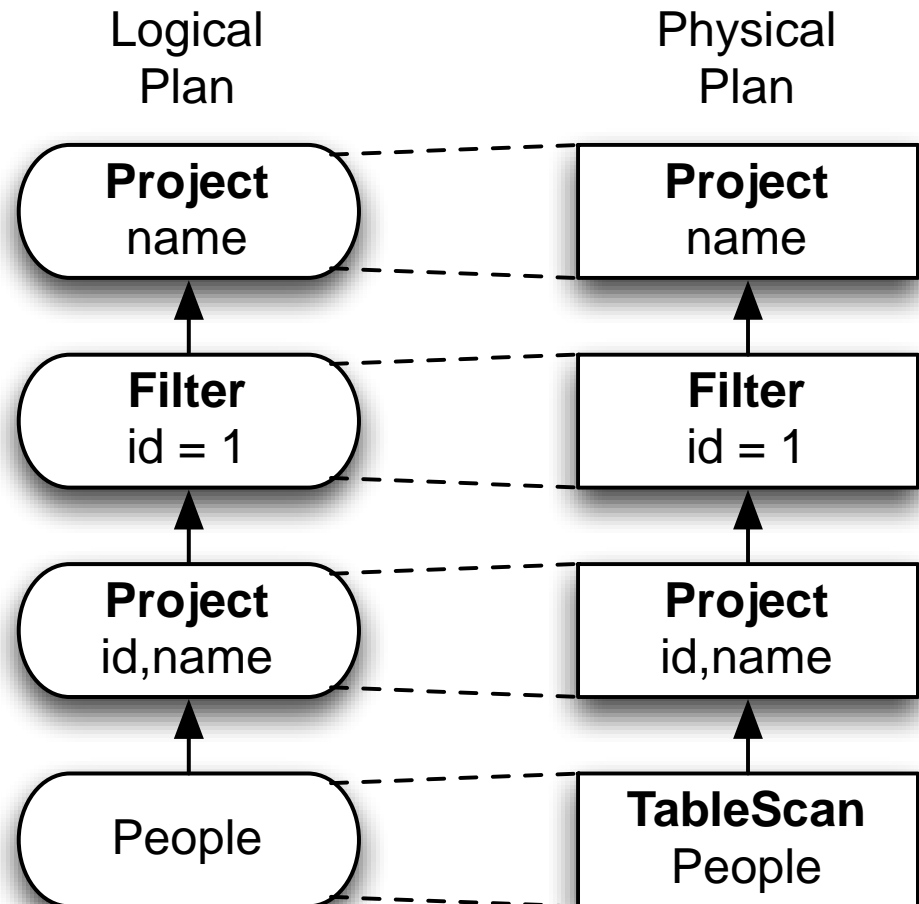
Etape ale planificării cererilor în Spark SQL. Dreptunghiurile rotunjite reprezintă arbori Catalyst

# Exemple de reguli de transformare



# Planuri logice vs. Planuri fizice

```
SELECT name
FROM (
  SELECT id, name
  FROM People) p
WHERE p.id = 1
```



# Bibliografie

1. Spark SQL, DataFrames and Datasets Guide  
(<https://spark.apache.org/docs/latest/sql-programming-guide.html>)
2. Catalyst Optimizer – Generic Logical Query Plan Optimizer  
(<https://jaceklaskowski.gitbooks.io/mastering-spark-sql/content/spark-sql-Optimizer.html>)
3. Databricks resources (<https://databricks.com/resources/slides>).