BIG DATA

CURS 2

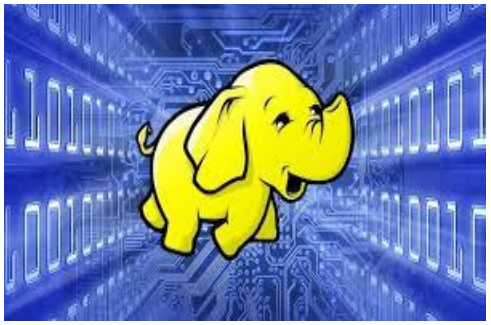
Apache Spark

Introducere

- Apache Spark este un framework open source pentru calcul în clustere (*cluster computing*)
- Inițial a fost dezvoltat la University of California, Berkeley's AMPLab, de către Matei Zaharia.
- Codul sursă a fost donat ulterior către Apache Software Foundation.
- Spark este un motor rapid pentru procesarea specifică big data.
- Generalizează modelul MapReduce, permițând mai multe tipuri de procesare.

De ce a fost necesară o nouă infrastructură?

- Hadoop a fost util în multe aplicații, cu cerințe și necesități diverse



- MapReduce a avut performanțe excelente în procesarea de tip *batch*, dar apoi s-au impus și alte cerințe:
  - Algoritmi mai complecși, *multi-pass*
  - Cereri ad-hoc mai interactive
  - Procesare de *stream-uri* mai apropiată de *real-time*

- Rezultatul: au apărut multe sisteme specializate pentru a răspunde acestor cerințe.

De ce a fost necesară o nouă infrastructură? Sisteme specializate

Sistemele Big Data:

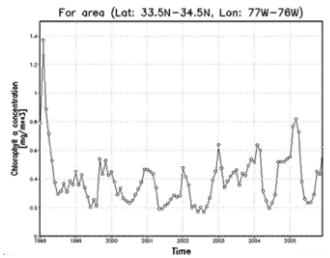
MapReduce → Pregel, Dremel, Giraph, Drill, Impala, Presto, Storm, S4, ... → Spark

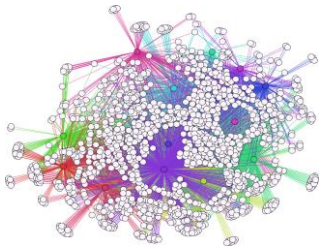
Procesare generală batch | **Sisteme specializate** | **Motor unificat**

Sisteme specializate: Dezavantaje

- Problemele sistemelor specializate:
  - Mai multe sisteme de gestionat, optimizat și desfășurat (*deploy*-at)
- Nu pot combina diferite tipuri de procesare în cadrul unei singure aplicații
  - Cu toate că multe execuții înlănțuite (*pipelines*) au nevoie de o astfel de funcționalitate
  - De exemplu: încărcare de date cu SQL, urmată de procesare *Machine Learning*.
- În multe *pipelines*, schimbul de date între motoare reprezintă, de fapt, costul dominant.

O aplicație poate avea nevoie de încărcare de date, procesare ML, vizualizare





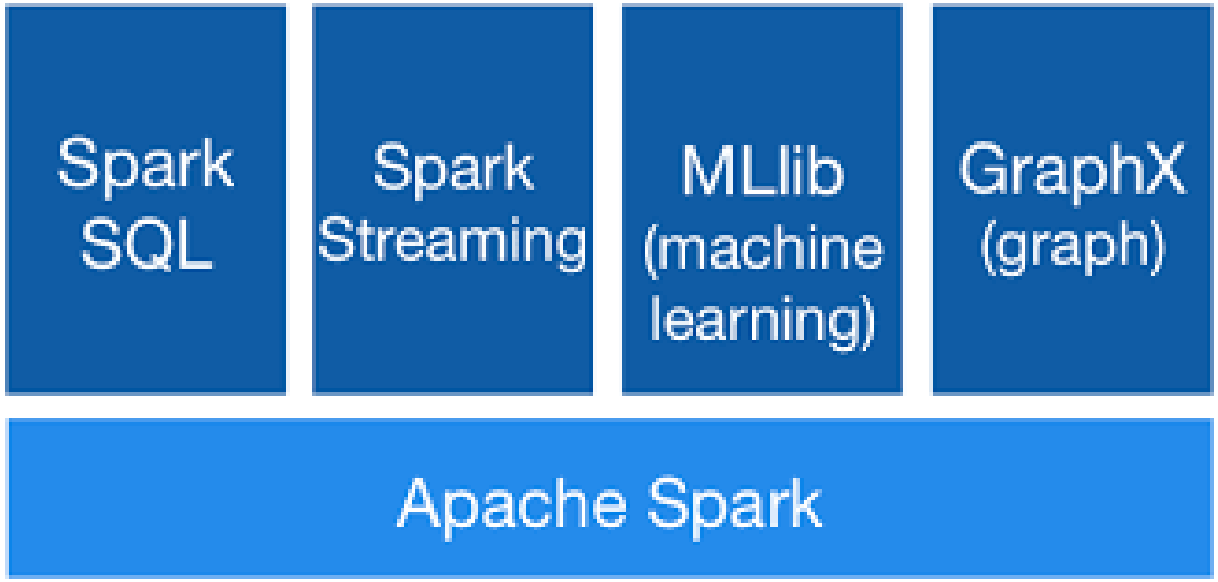
Viziunea: Infrastructură **generică eficientă**

Sistemele Big Data:

MapReduce → Pregel, Dremel, Giraph, Drill, Impala, Presto, Storm, S4, … → Spark

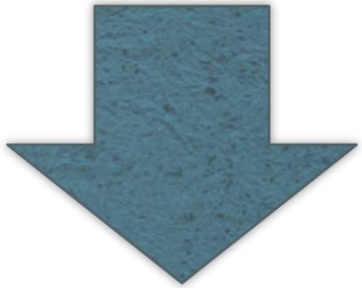
Procesare generală batch | **Sisteme specializate** | **Motor unificat**

Viziunea: Infrastructură **generică eficientă**



Cerințele care au motivat noua infrastructură

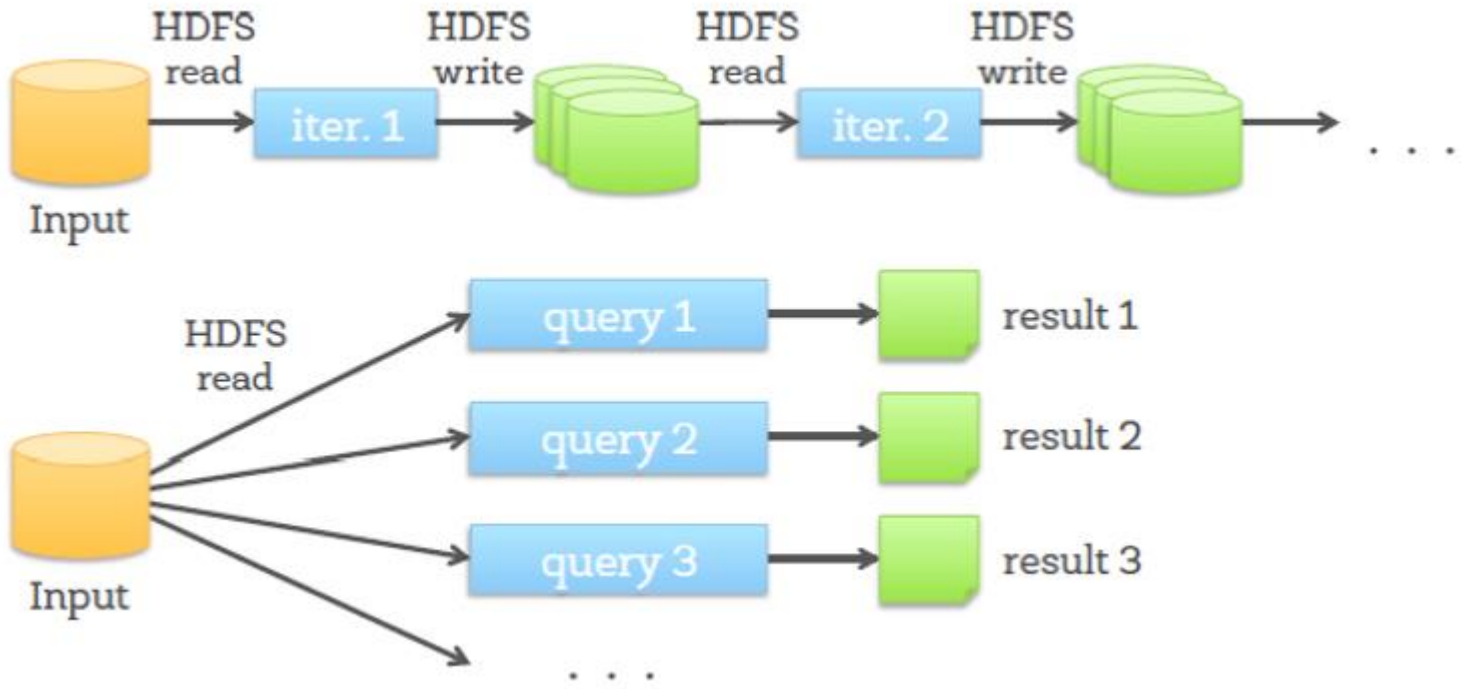
- Algoritmi complecși *multi-pass*
- Cereri interactive ad-hoc
- Procesare *real-time* de stream-uri



Toate acestea necesită partajare și transfer eficient de date -> lucruri ce lipsesc în MapReduce

Motivarea apariției unei noi infrastructuri
Pornind de la …

Partajarea datelor în MapReduce

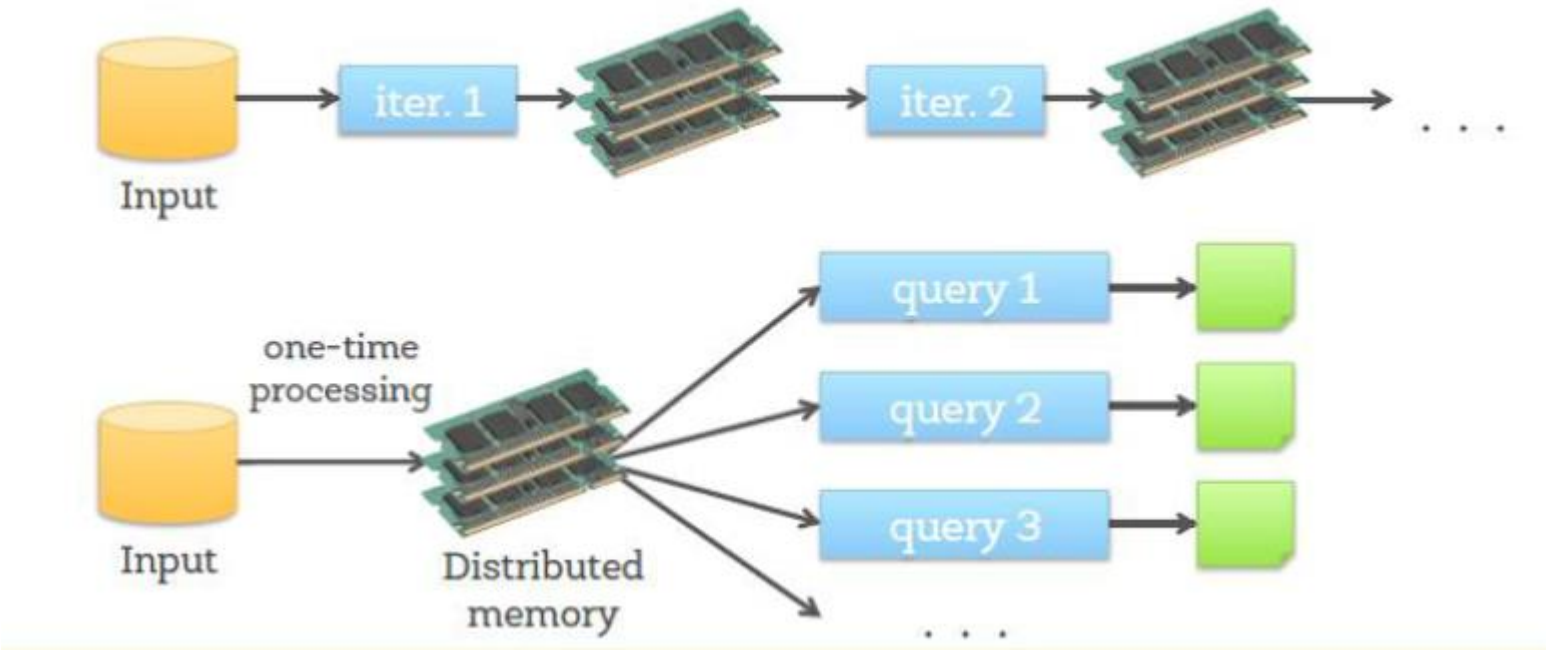


Lentă din cauza replicării datelor și a operațiilor I/O

Motivarea apariției unei noi infrastructuri

Până la …

Ce ar fi necesar?



De 10-100 ori mai rapid decât rețeaua și discul

Motivarea apariției unei noi infrastructuri
Pornind de la …

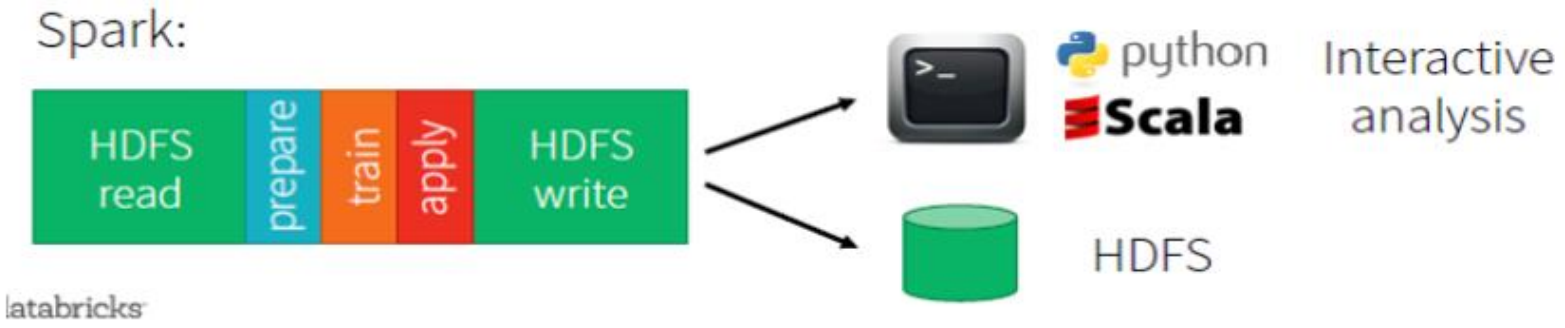
Fluxul de execuție

Motoare separate:

Motivarea apariției unei noi infrastructuri

Până la…

Fluxul de execuție



Motivarea din punct de vedere *hardware*

- Memoria RAM a devenit tot mai ieftină
- Calculatoarele au memorie RAM de ordinul GBs
- Există o cantitate mare de RAM distribuit în cluster

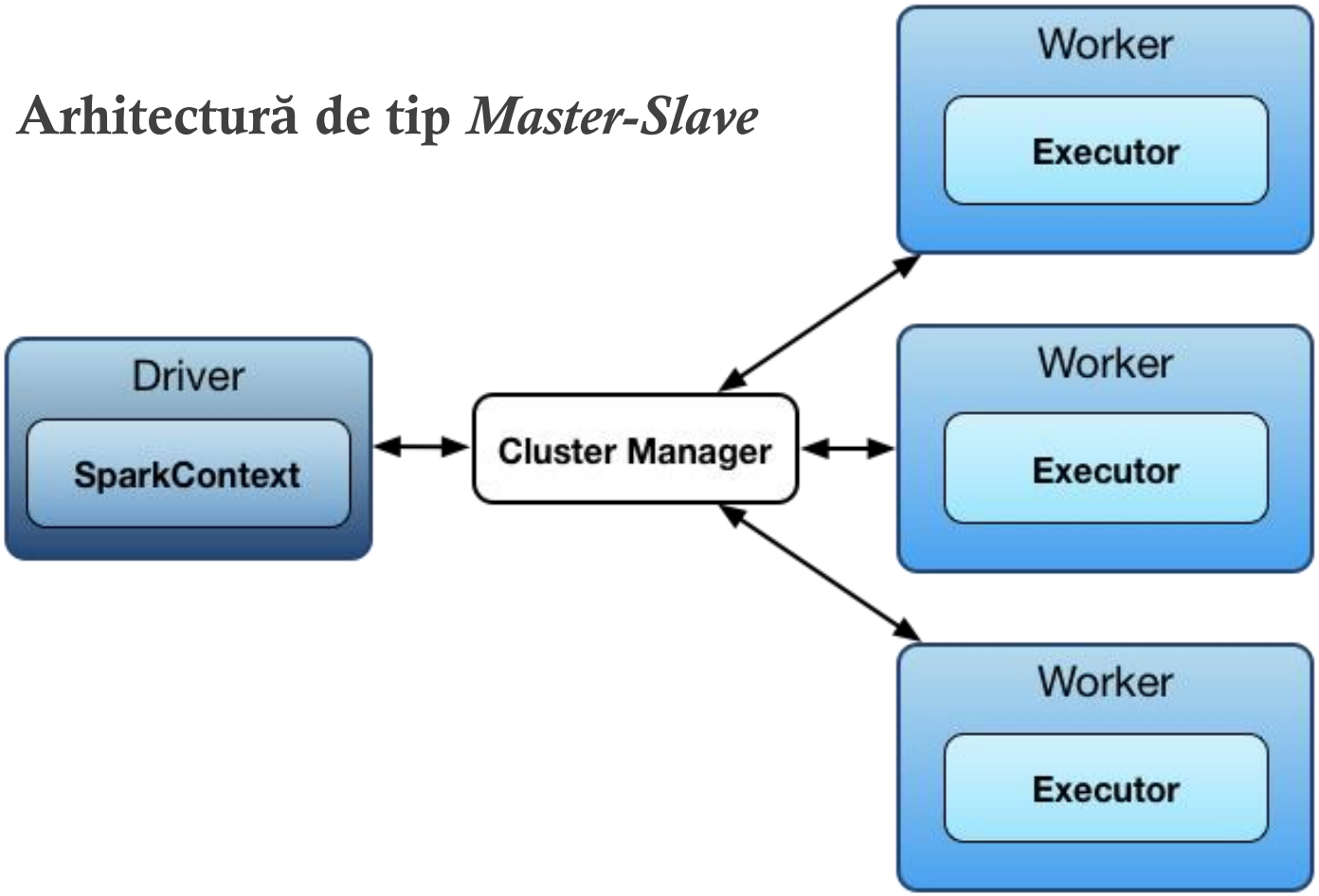
O mare parte din procesare, stocare și transfer de date ar trebui să folosească memoria RAM

Motivarea infrastructurii: Rezumat

- **Suport mai bun pentru procesarea real-time**
- **Exploatarea cât mai mult posibil a memoriei RAM**
- **Distribuire *large-scale***
- **Adaptarea conceptelor și tehnicilor anterioare**

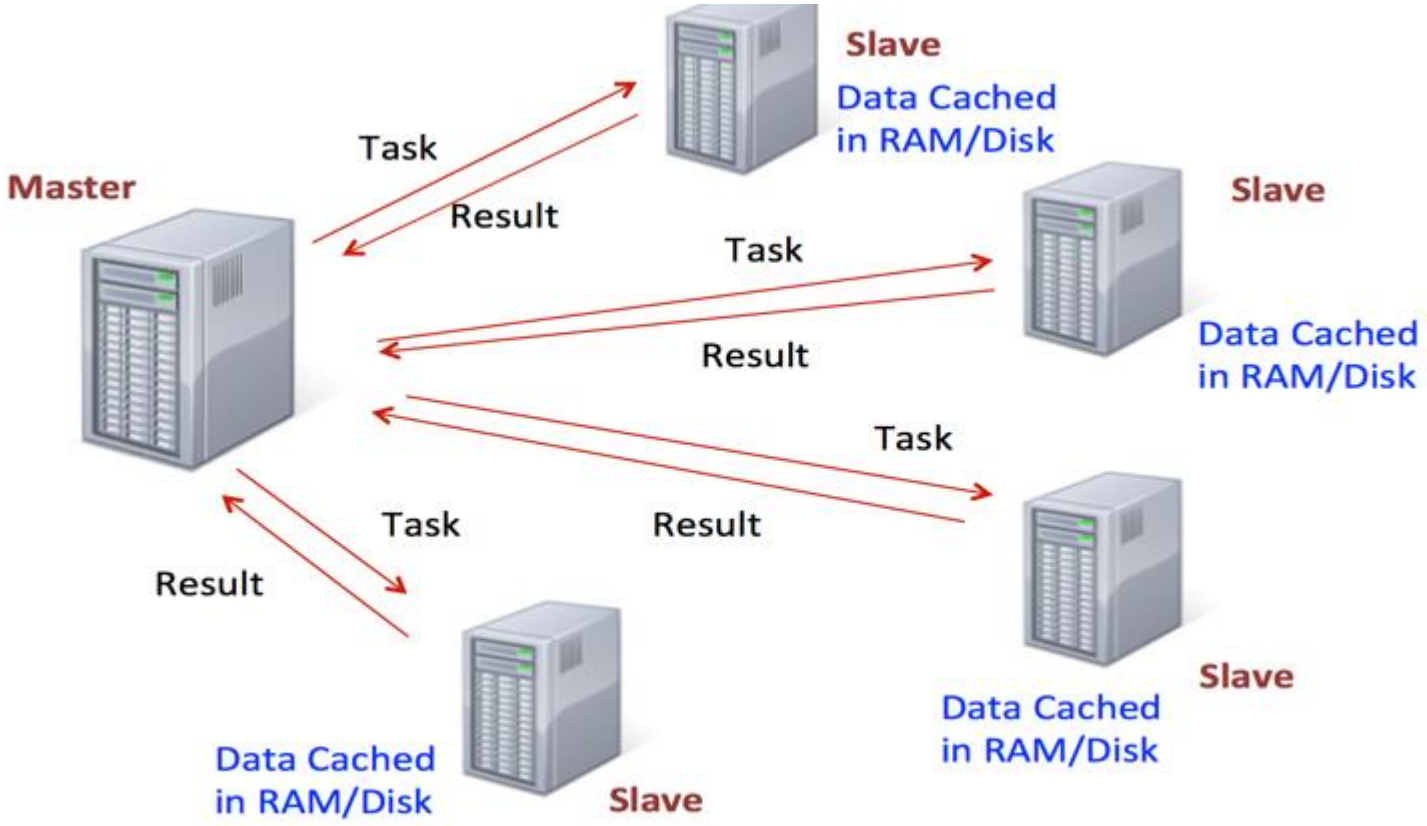
Arhitectura Spark

- **Arhitectură de tip *Master-Slave***

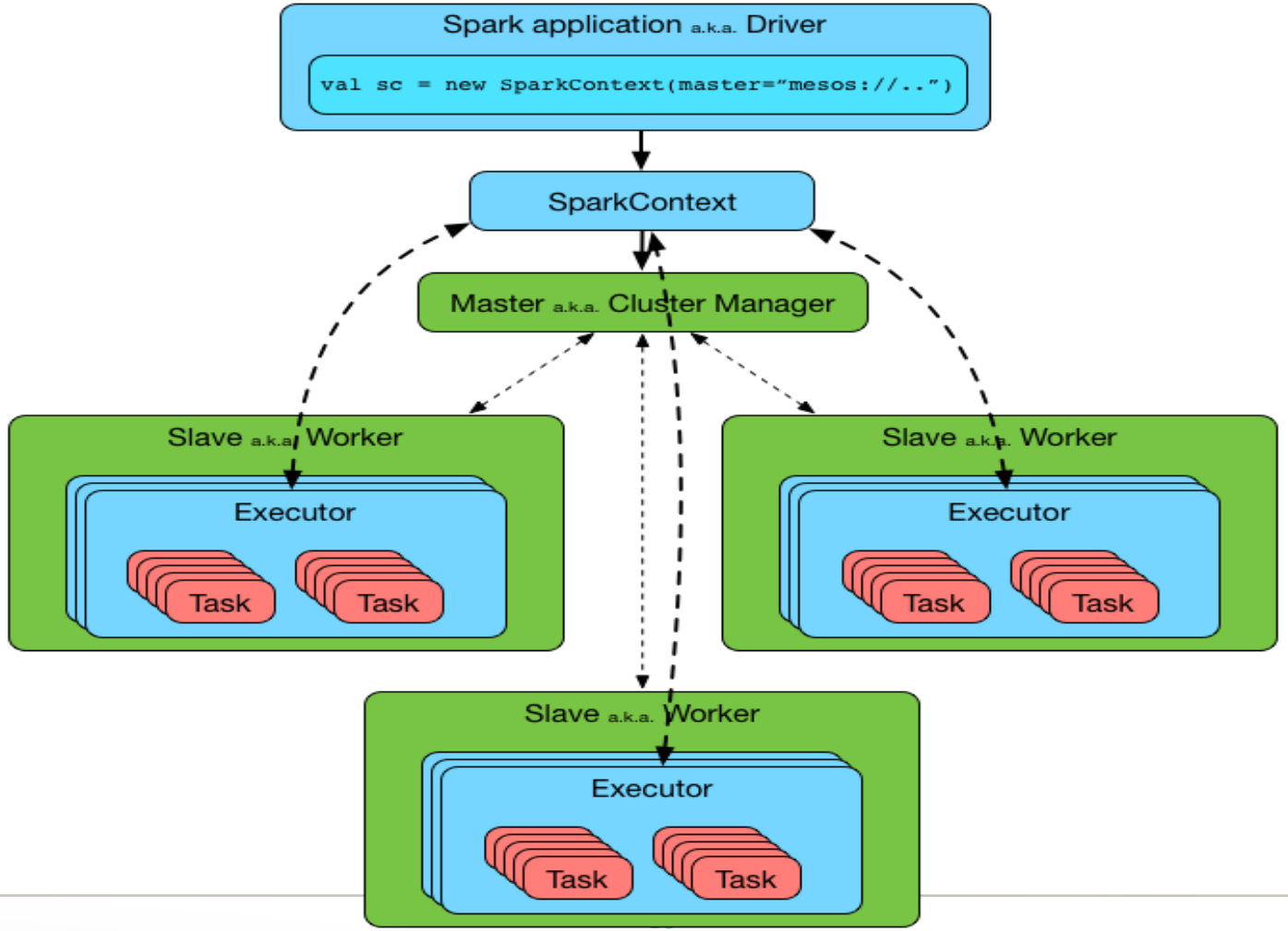


Modelul de comunicare din Spark

Cum execută Spark un job?

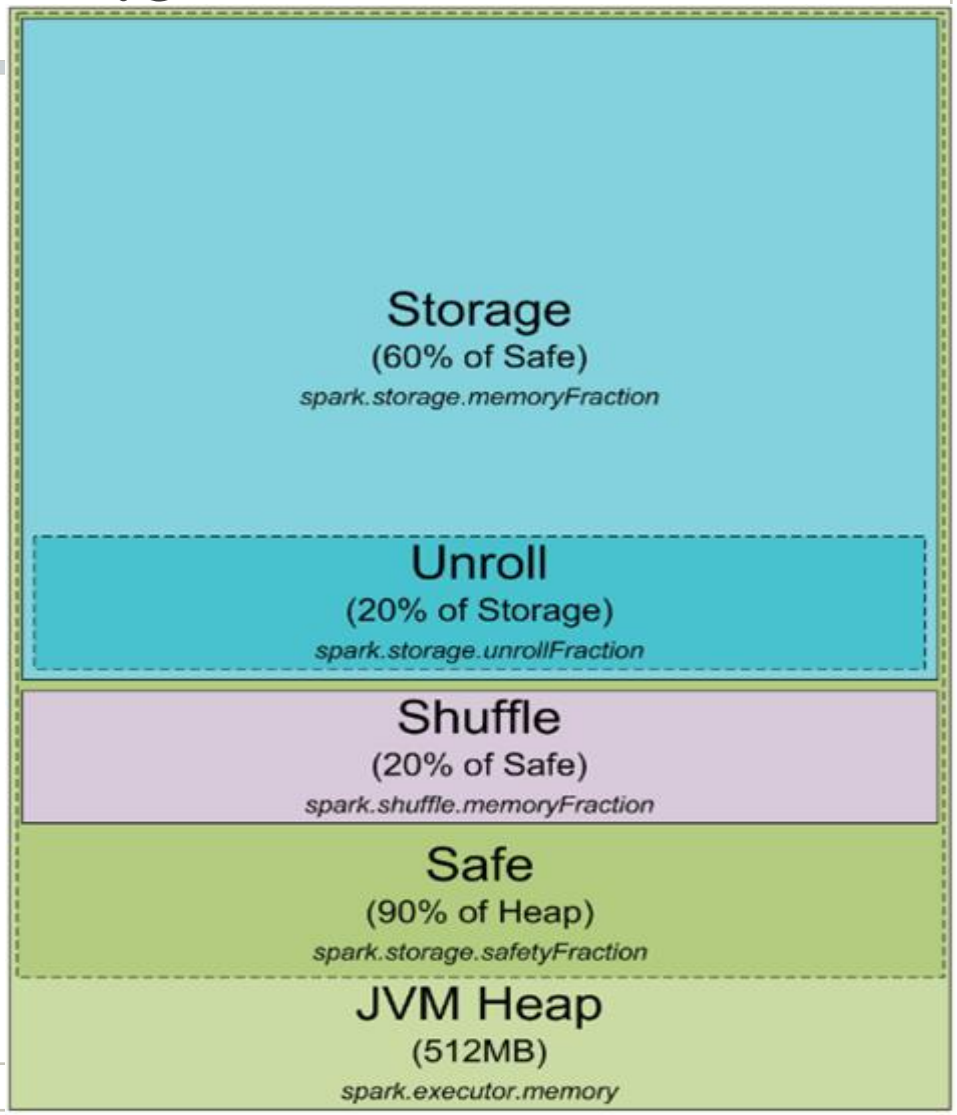


Exemplu [Link]

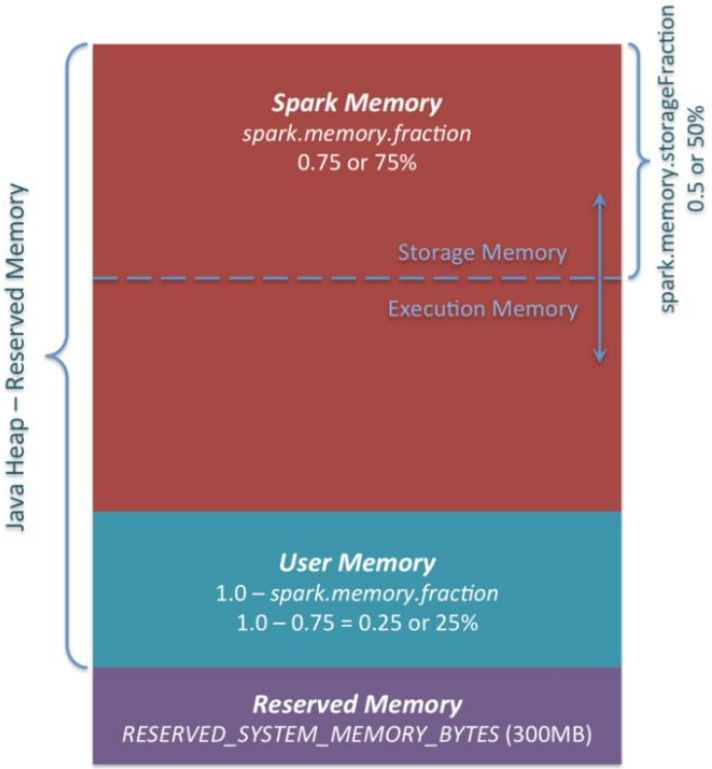


Gestiunea memoriei în Spark <=1.5

- Utilizarea memoriei este esențială în Spark (*Caching*)
- Procesul Spark este un proces JVM



Gestiunea memoriei în Spark 1.6.0+



Modelul de programare din Spark

1. Dezvoltatorii scriu **programe *driver*** care implementează, la nivel înalt, fluxul de control al aplicației lor și lansează diferite operații în paralel.
   - Scriere de cod de nivel înalt pentru a construi un workflow (Scala/Python/Java)
   - Codul este compilat în operații paralele distribuite
2. Spark furnizează 2 abstractizări principale pentru programarea paralelă:
   - **RDDs: Resilient Distributed Datasets**
   - **Operațiile paralele pe aceste dataset-uri**
3. Spark suportă 2 tipuri de variabile partajate ce pot fi folosite în cadrul funcţiilor ce rulează în cluster-e.

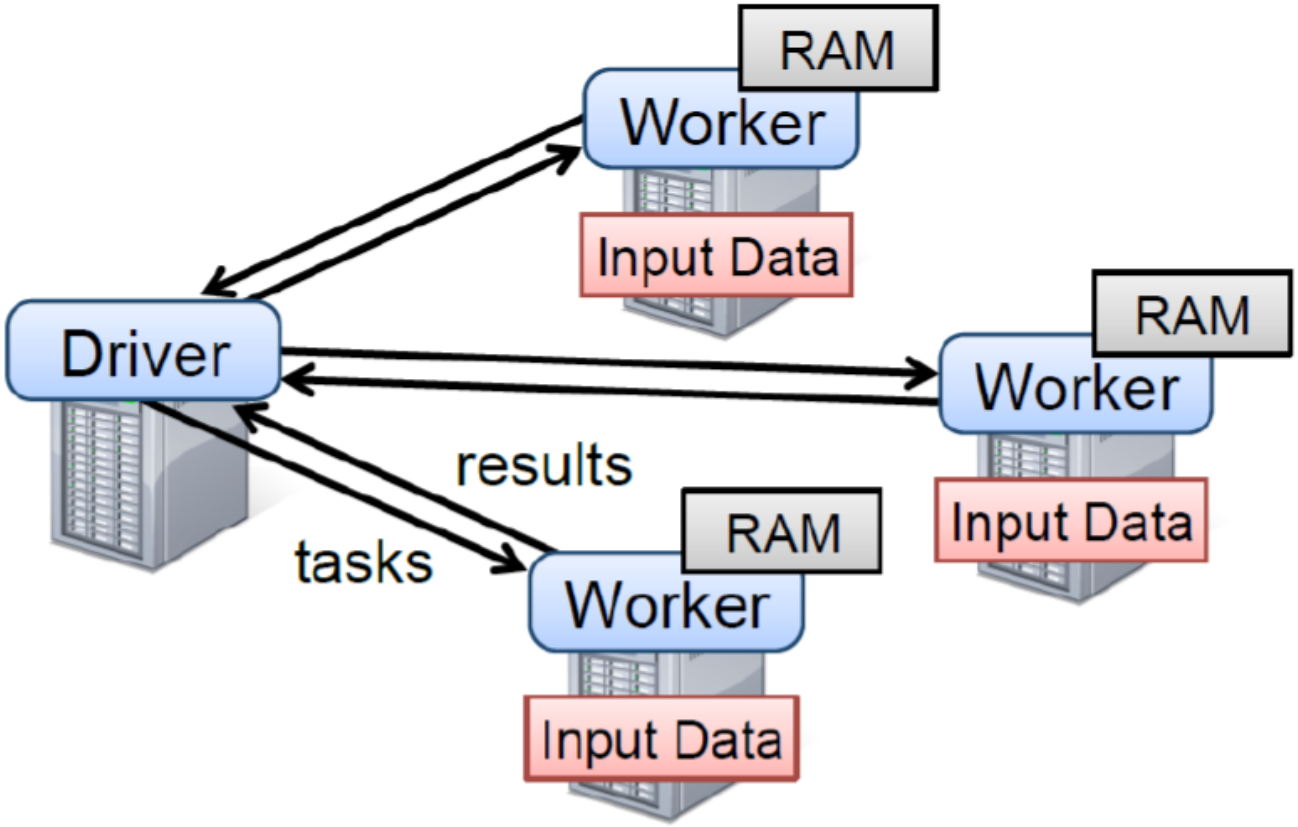
Modelul de programare din Spark

Dezvoltatorii pot crea două tipuri de variabile partajate ce permit două pattern-uri de utilizare (simple, dar comune):

- Variabile *Broadcast*: Dacă anumite date *read-only*, de dimensiuni foarte mari, sunt folosite în operaţii paralele multiple, este preferabilă distribuirea acestora în *worker*-i o singură dată (în loc de a fi transmise în cadrul fiecărui *task*). Variabilele *broadcast* permit acest lucru. Acest mecanism este utilizat şi automat de către Spark - realizează *broadcast*-ul datelor comune necesare *task*-urilor din fiecare etapă. Datele *broadcasted* sunt reţinute în *cache* în formă serializată, iar înainte de execuţia *task*-ului sunt deserializate.

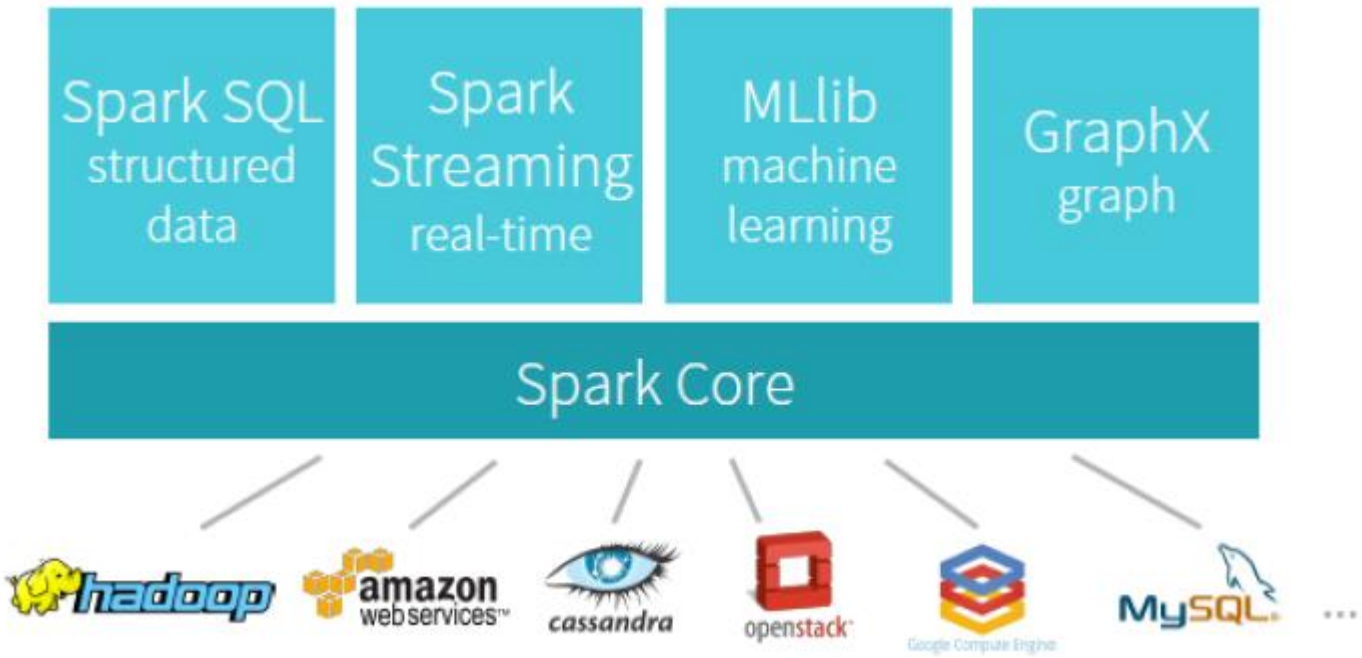
- Acumulatori: Acestea sunt variabile pe care *worker*-ii doar le pot adăuga utilizării unei operaţii asociative şi pe care doar *driver*-ul le poate citi. Sunt utile în calculele totalizatoare paralele şi sunt rezistente la erori.

Modelul de programare din Spark



Modelul de programare din Spark

Un motor general



- Limbaj de programare
- Combină programarea orientată pe obiecte cu cea funcțională
- Compilează în *bytecode* Java
- Rulează pe JVM

Spark RDDs

RDD: Concept

Resilient Distributed Datasets

- Colecție de obiecte (înregistrări) care se comportă ca o singură unitate
- Stocare în memoria principală sau pe disc, în cadrul unui cluster
- Se pot construi prin operații paralele
- Operațiile paralele se efectuează pe RDD
- Asigură rezistența la erori (*fault tolerance*) fără replicare (prin descendență – *lineage*)

RDD: Concept

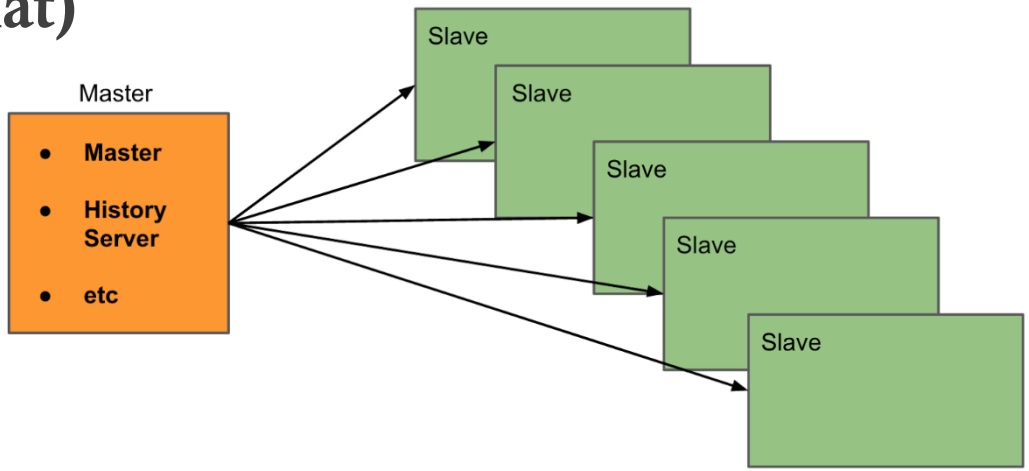
Exemple de operații paralele ce pot fi efectuate pe RDD-uri:

- *reduce*: combină elemente ale setului de date folosind o funcție asociativă pentru a produce un rezultat în programul *driver*.
- *collect*: trimite toate elementele setului de date către programul *driver*.
- *foreach*: aplică o funcție definită de utilizator pe fiecare element din RDD.

RDD: Concept

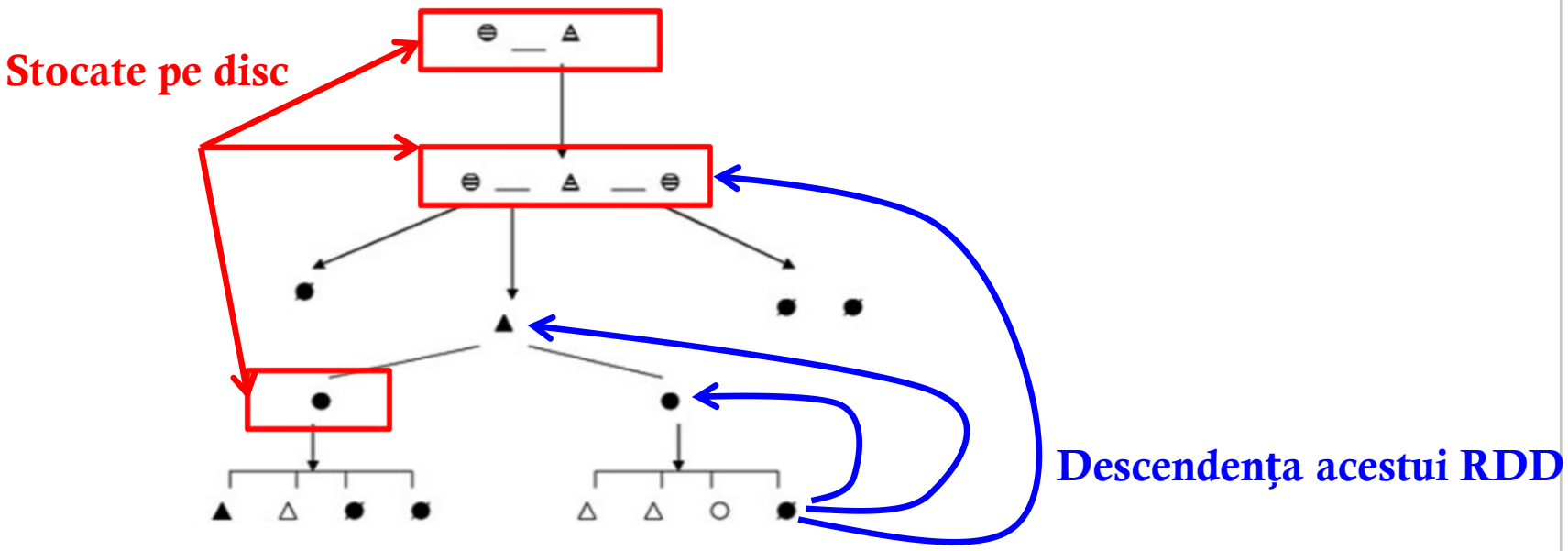
- **Un RDD este *read-only***

- **Distribuirea are loc fie în memoria principală, fie pe disc (se decide automat)**



RDD: *Fault Tolerance*

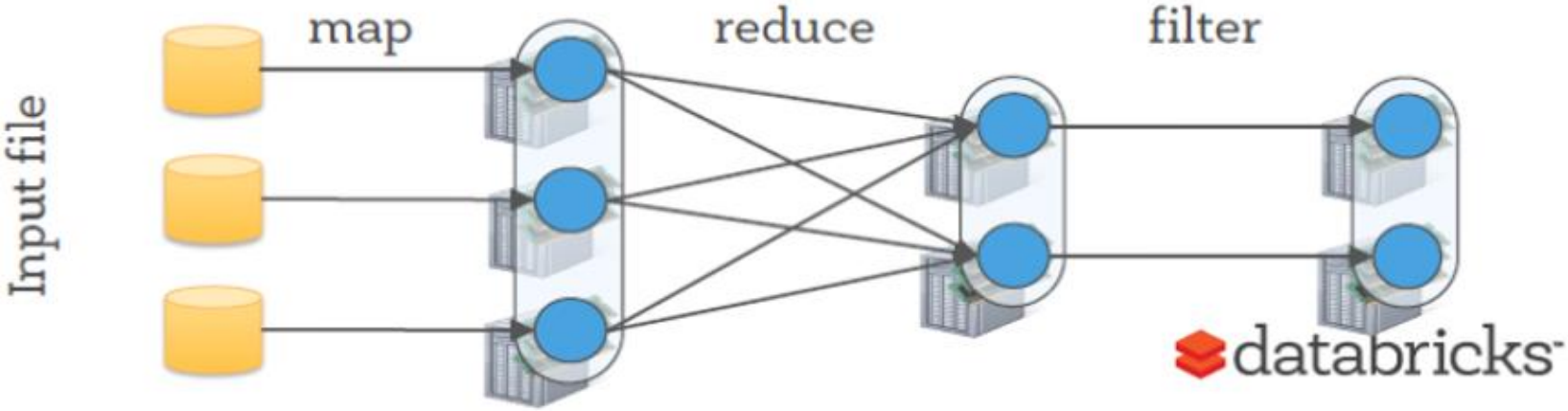
- Datele nu trebuie replicate
- Trebuie menținută descendența (proveniența) pentru a le putea re-crea pornind de la date dintr-o unitate de stocare fiabilă (de încredere)



RDD: *Fault Tolerance*

- RDD-urile mențin informația legată de proveniență pentru a putea reconstrui date pierdute

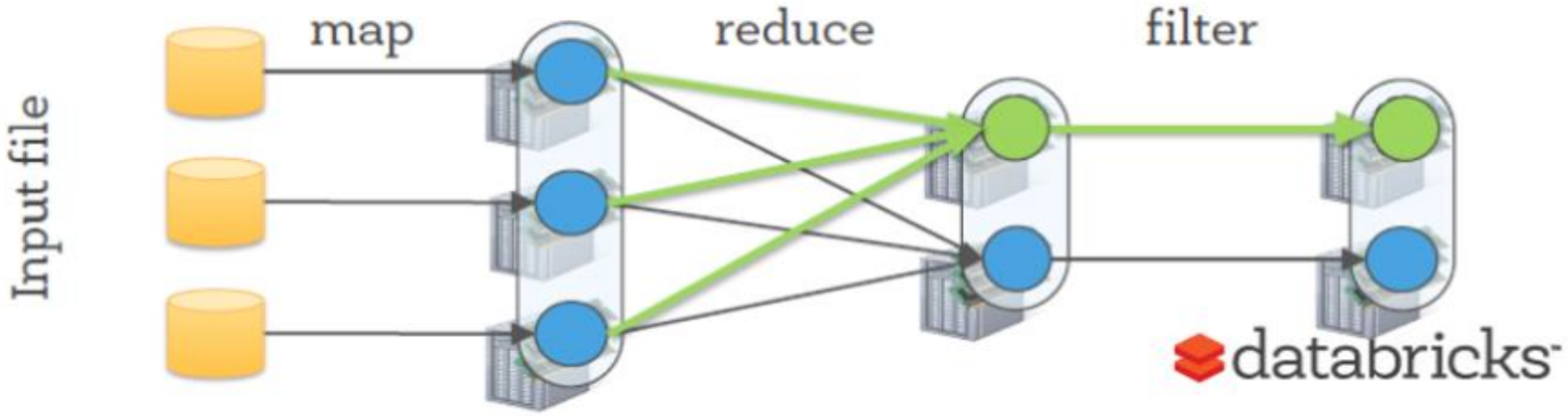
```
file.map(lambda rec: (rec.type, 1))
    .reduceByKey(lambda x, y: x + y)
    .filter(lambda (type, count): count > 10)
```



RDD: *Fault Tolerance*

- RDD-urile mențin informația legată de proveniența pentru a putea reconstrui date pierdute

```
file.map(lambda rec: (rec.type, 1))
    .reduceByKey(lambda x, y: x + y)
    .filter(lambda (type, count): count > 10)
```



RDD: Controlul din partea utilizatorilor

- Strategii de persistență și partiționare
- Indică ce RDD-uri vor refolosi și aleg stategii de stocare pentru acestea (de exemplu, stocare *in-memory*)
- Solicită ca un RDD să fie partiționat pe mai multe mașini – util pentru optimizarea plasării datelor.

RDD: Avantaje

- MapReduce accesează puterea computațională a clusterului, dar nu și memoria distribuită
  - Lent, consumator de timp, inficient pentru aplicațiile ce reutilizează rezultatele intermediare.

- RDD-urile permit stocarea datelor intermediare *in-memory*, permițând reutilizarea eficientă a datelor.

RDD vs. Modelul tradiţional de partajare a memoriei

Aspect	RDD	Memorie partajată distribuită
Citire	Granularitate mare sau fină	Granularitate fină
Scriere	Granularitate mare	Granularitate fină
Consistenţă	Trivial (imutabil)	Depinde de aplicaţie / runtime
Recuperarea după defecte (*fault recovery*)	Granularitate fină şi cost suplimentar (*overhead*) scăzut folosind provenienţa (*lineage*)	Necesită salvarea stărilor (crearea de *checkpoints*) şi *rollback* în program
Atenuarea dispersiei	Posibil cu ajutorul *task*-urilor de *backup*	Dificil de realizat
Plasarea procesării	Automat, bazat pe localizarea datelor	Depinde de aplicaţie
Comportament în caz că nu există suficient RAM	Similar sistemelor existente de fluxuri de date	Performanţă slabă (memory swapping?)

Crearea RDD-urilor

- Încărcare din dataset extern (fișier)
- Creare din alt RDD (transformare)
- Paralelizarea unei colecții centralizate

Crearea RDD-urilor

1. **Încărcarea unui dataset extern**

- Cea mai frecventă metodă de creare a unui RDD
- Datele pot fi localizate în orice sistem de stocare (precum HDFS, Hbase, Cassandra etc.)
- Exemplu:

```
lines = spark.textFile("hdfs://...")
```

Support for HDFS, HBase, Amazon S3, …

RDD: #partitions = #of HDFS blocks

Crearea RDD-urilor

2. Crearea unui RDD dintr-un RDD existent

- Un RDD existent poate fi utilizat pentru a crea un RDD nou
- RDD-ul părinte rămâne intact (nu este modificat)
- RDD-ul părinte poate fi utilizat în continuare pentru alte operații
- Exemplu:

```
errors = lines.filter(_.startsWith("ERROR"))
```

New RDD

Crearea RDD-urilor

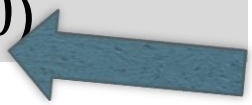
3. Paralelizarea unei colecții centralizate

```
val data = Array(1, 2, 3, 4, 5, 100, 8, 7, ….)

val distData = sc.parallelize(data)
```

```
val data = Array(1, 2, 3, 4, 5, 100, 8, 7, ….)

val distData = sc.parallelize(data, 10)
```



Crearea a 10 partiții

Operații asupra RDD-urilor

Crearea unui nou RDD

Pentru aceste operații nu este declanșată nicio execuție

Operații de tip transformare (Transformation Ops.)

Similare părții *map* din Hadoop

&

Returnarea unei valori către apelant

Pentru aceste operații este declanșată o execuție

Operații de tip acțiune (Action Ops.)

Similare părții *reduce* din Hadoop

Operații de tip transformare

- Operează asupra unui RDD și generează un alt RDD
- Evaluare de tip *lazy*
- RDD-ul de intrare rămâne intact
- Exemple: ***map***, ***filter***, ***join***
  - Rularea unui filtru pe un RDD produce un alt RDD.

Operații de tip transformare: Exemplul 1

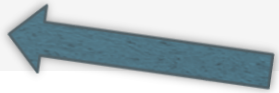
Transformări

```
lines = spark.textFile("hdfs://...")
errors = lines.filter(_.startsWith("ERROR"))
```

- RDD-ul original (părinte) rămâne intact și poate fi folosit în viitor pentru alte transformări.
- Nu are loc nicio acțiune, sunt create doar metadatele RDD-ului `errors`.

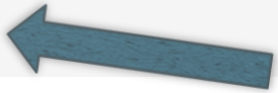
Operații de tip transformare: Exemplul 2

```
val lines = sc.textFile("data.txt")
val lineLengths = lines.map(s => s.length)
```



La latitudinea lui Spark dacă îl păstrează în memorie sau îl recalculează atunci când este necesar

```
lineLengths.persist()
```



Solicităm păstrarea acestui RDD în memorie

Operații de tip acțiune

- O acţiune este o operație ce efectuează un calcul pe un RDD existent și produce un rezultat.
- Rezultatul poate fi:
  - Returnat programului *Driver*.
  - Stocat într-un sistem de fișiere (precum HDFS)
- Exemple:
  - *count*()
  - *collect*()
  - *reduce*()
  - *save*()

Operații de tip acțiune: Exemplul 1

```
val lines = sc.textFile("data.txt")
val lineLengths = lines.map(s => s.length)
val totalLength = lineLengths.reduce((a, b) => a + b)
```

Operații de tip acțiune: Exemplul 2 [Link]

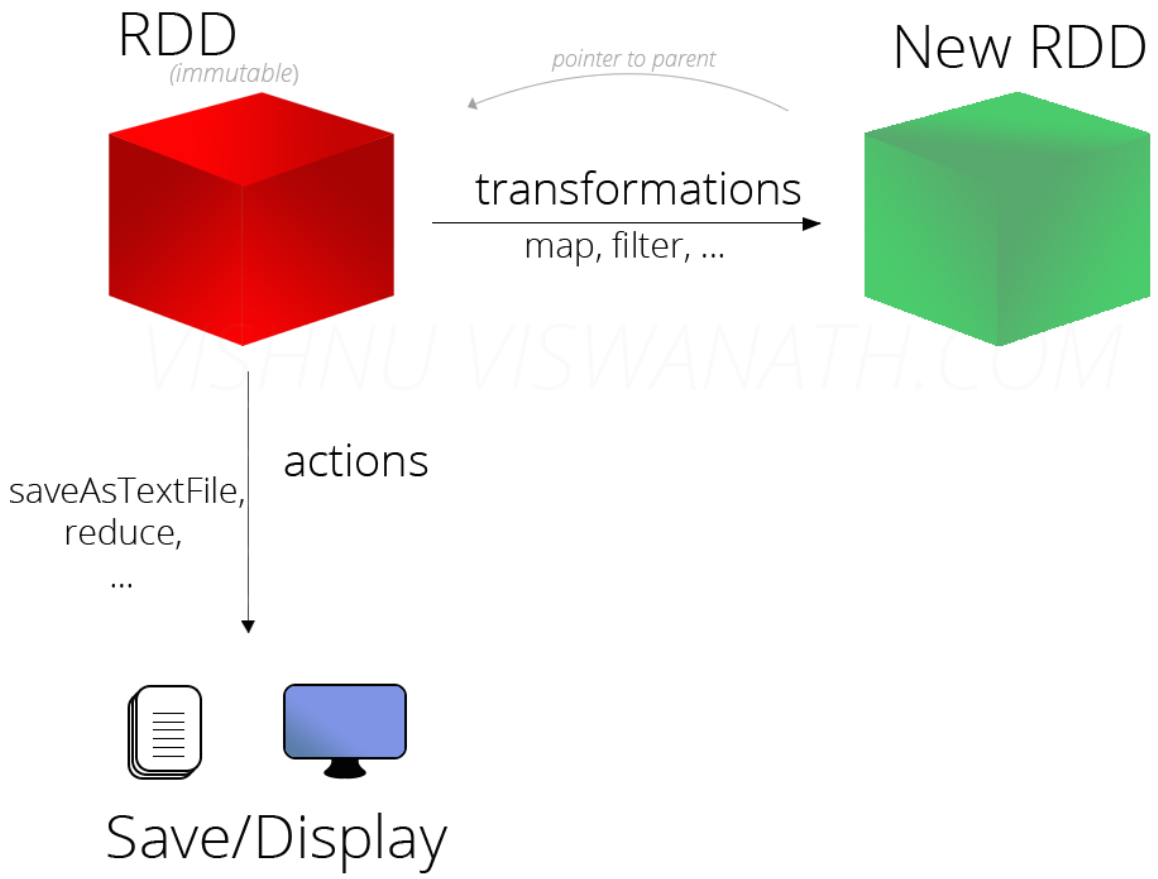
```
val logFile = "hdfs://master.backtobazics.com:9000/user/root/sample.txt"
val lineRDD = sc.textFile(logFile)
//Transformation 1 -> DAG created
//{DAG: Start -> [sc.textFile(logFile)]}

val wordRDD = lineRDD.flatMap(_.split(" "))
//Transformation 2 -> wordRDD DAG updated
//{DAG: Start -> [sc.textFile(logFile)]
//               -> [lineRDD.flatMap(_.split(" "))]}

val filteredWordRDD = wordRDD.filter(_.equalsIgnoreCase("the"))
//Transformation 3 -> filteredWordRDD DAG updated
//{DAG: Start -> [sc.textFile(logFile)]
//               -> [lineRDD.flatMap(_.split(" "))]
//                  -> [wordRDD.filter(_.equalsIgnoreCase("the"))]}


filteredWordRDD.collect
//Action: collect
//Execute DAG & collect result to driver node
```

Transformări vs. Acțiuni



Evaluarea întârziată (*lazy evaluation*)

- Operațiile de transformare pe RDD-uri sunt evaluate cu întârziere (*lazy*)
- Rezultatele nu sunt calculate efectiv (fizic) imediat
- Sunt înregistrate metadatele referitoare la transformări
- Transformările sunt implementate doar atunci când este invocată o acțiune

Exemplu

```
lines = spark.textFile("hdfs://...")

errors = lines.filter(_.startsWith("ERROR"))

errors.count()
```

Aici se declanșează execuția

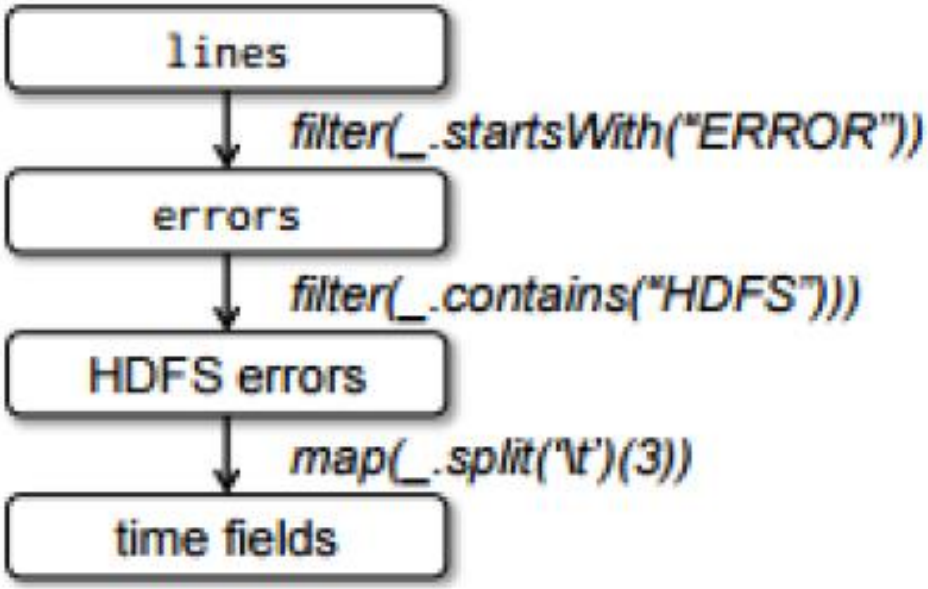
- RDD-ul `errors` nu este returnat programului *driver* până când o acțiune este invocată asupra acestui RDD.

Toleranța la erori a RDD (*Fault Tolerance*)

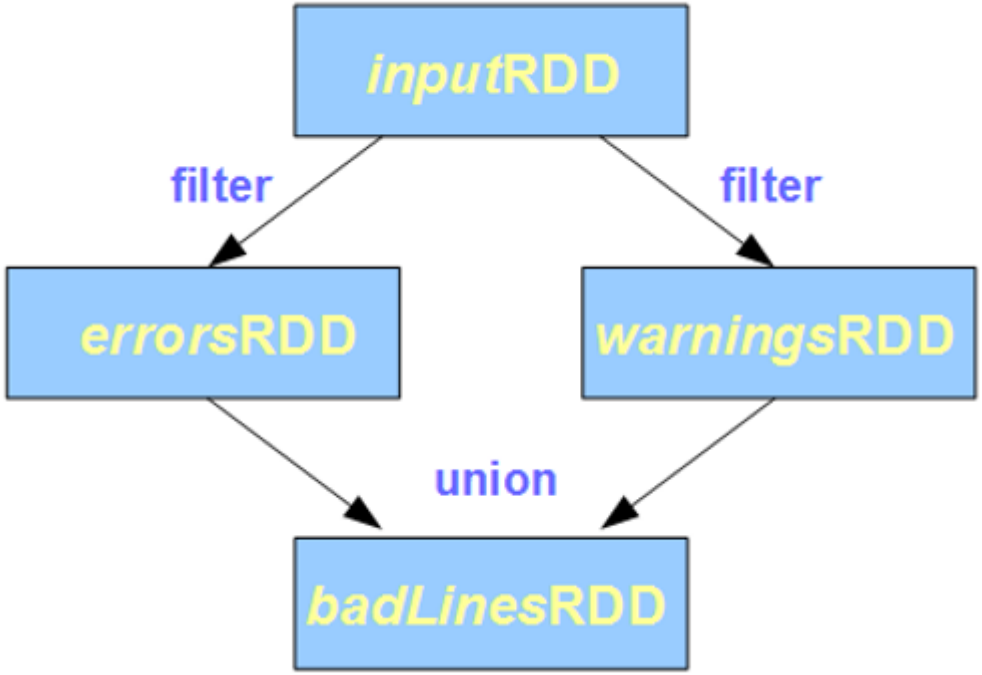
- RDD-urile *in-memory* nu sunt replicate
  - Memoria RAM este totuși o **resursă limitată**
- Dacă un nod eșuează, operațiile pot continua
- Abordarea Spark: **Graful de descendență (*Lineage Graph*)**
  - Este un graf orientat aciclic (Directed Acyclic Graph – DAG)
- Sunt menținute dependențele între RDD-uri
- Permit revenirea la cel mai apropiat RDD bazat pe disc (*disk-based* RDD)
- Grafurile DAG sunt menținute de către DAGScheduler

Graful de descendență (*Lineage Graph*)

- Model care descrie pașii necesari pentru a crea rezultatul procesului de transformare.
- Nu sunt stocate datele, ci modul în care acestea au fost generate (pașii de procesare)



Lineage Graph

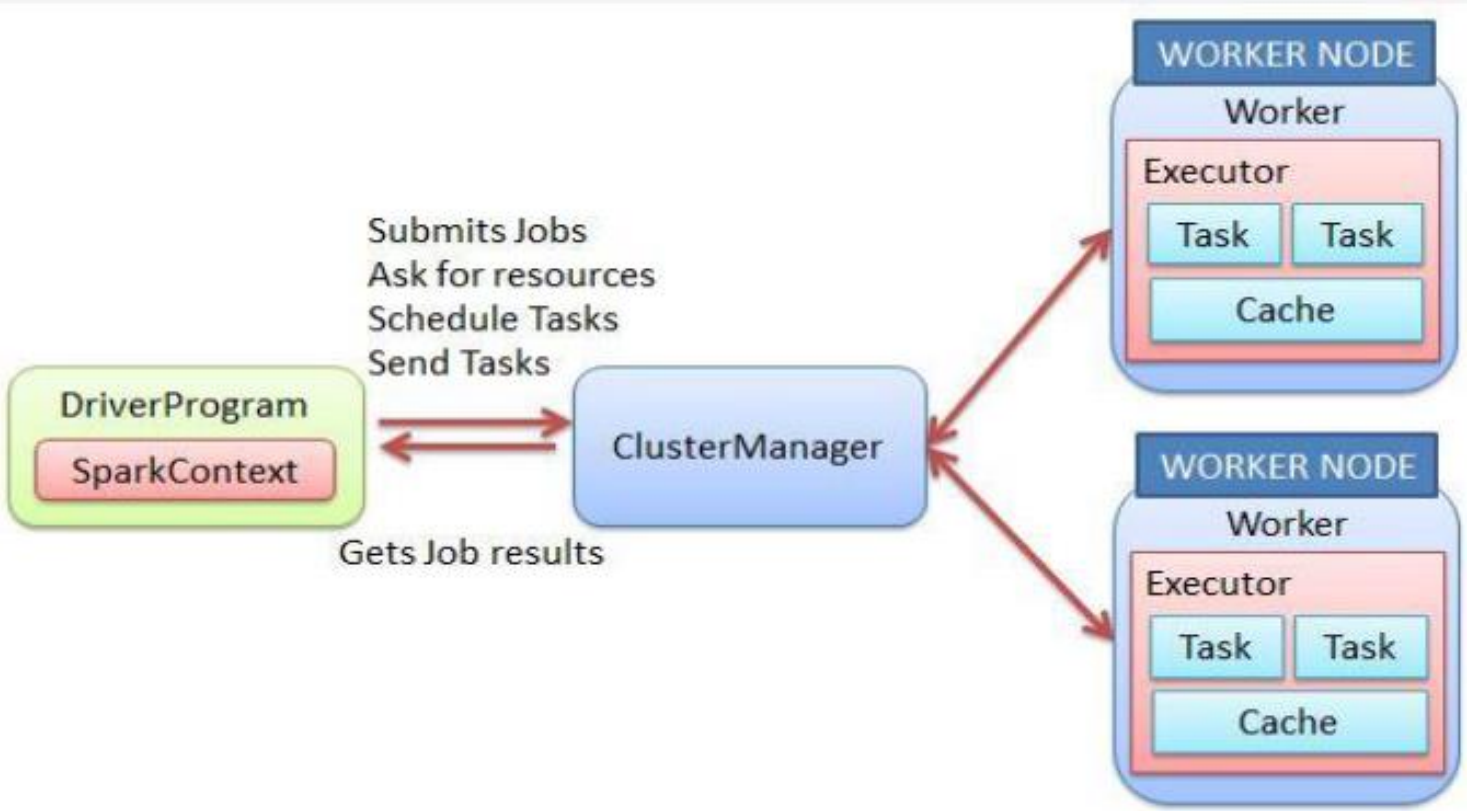


Graf de descendență creat în cadrul analizei log-urilor

Aplicații neadecvate pentru RDD-uri

- RDD-urile sunt potrivite pentru aplicațiile batch care aplică aceeași operație pe toate elementele unui set de date
- Mai puțin potrivite pentru aplicațiile cu actualizări asincrone de tip fine grained pe o stare partajată
  - Sistem de stocare pentru aplicații web
  - Web crawler incremental

Spark Programming Interface



Spark Programming Interface

Programul *driver*

- Fiecare aplicație Spark are un *driver program*
- Acesta este responsabil cu lansarea task-urilor paralele în diferitele noduri ale cluster-ului
- Încapsulează funcția main() a codului
- Definește seturi de date distribuite în noduri
- Aplică operațiile necesare în seturile de date distribuite

Spark Programming Interface

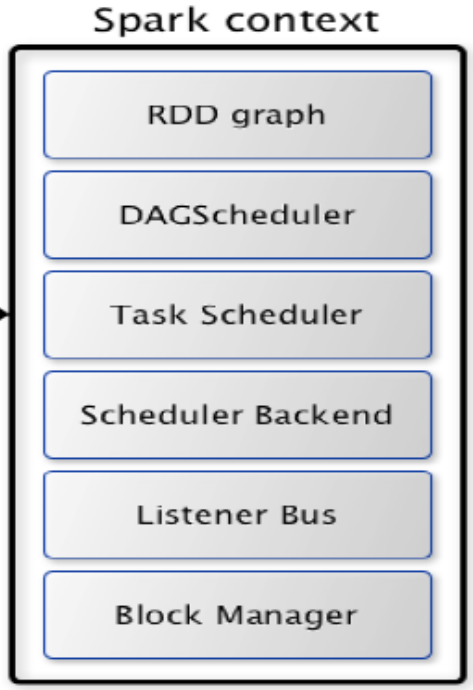
SparkContext

- Mijloc de conectare a *driver program* la cluster
- Odată ce SparkContext este pregătit, el poate fi utilizat pentru a construi un RDD

```
val sc = new SparkContext(master="local[*]",
            appName="SparkMe App", new SparkConf)

val lines = sc.textFile(...).cache()

val c = lines.count()
println(s"There are $c lines in $fileName")
```



Spark Programming Interface

Executori

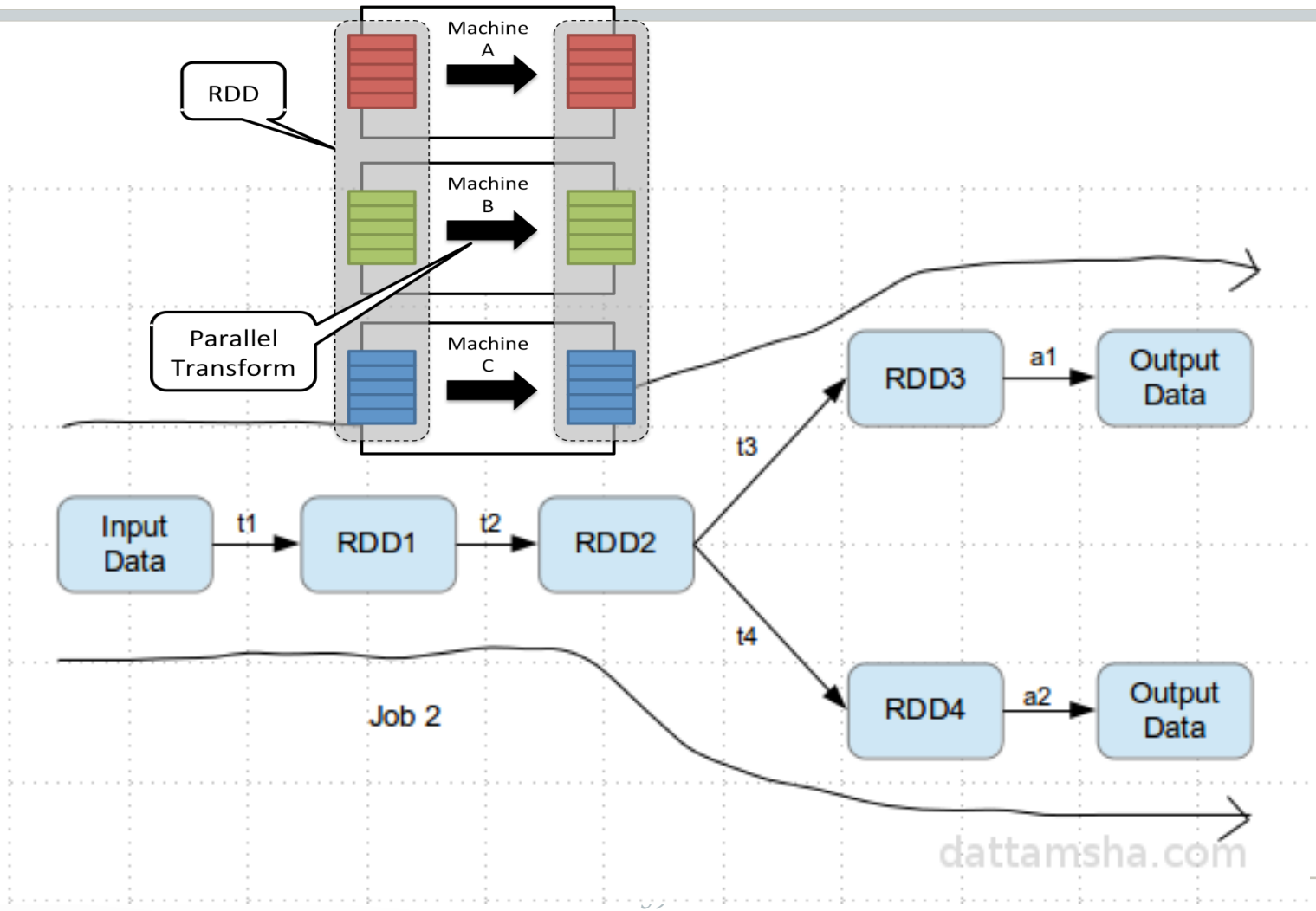
- *Driver program* gestionează nodurile numite executori
  - Aceste noduri sunt utilizate pentru rularea operațiilor distribuite
  - Fiecare executor efectuează o parte din operație

- Exemplu:
  - Câte o partiție diferită de date este trimisă fiecărui executor
  - Fiecare executor determină numărul de linii din partiția lui de date

Reprezentarea RDD-urilor

- **Fiecare RDD se împarte în:**
  - Partiții multiple
  - Dependențe de RDD-ul (sau RDD-urile) părinte
- **Cum pot fi reprezentate dependențele între RDD-uri?**
- **Există 2 tipuri de dependențe:**
  - Strânse (*Narrow*)
    - Exemplu: *map*
  - Largi (*Wide*)
    - Exemplu: *join*

Reprezentarea RDD-urilor

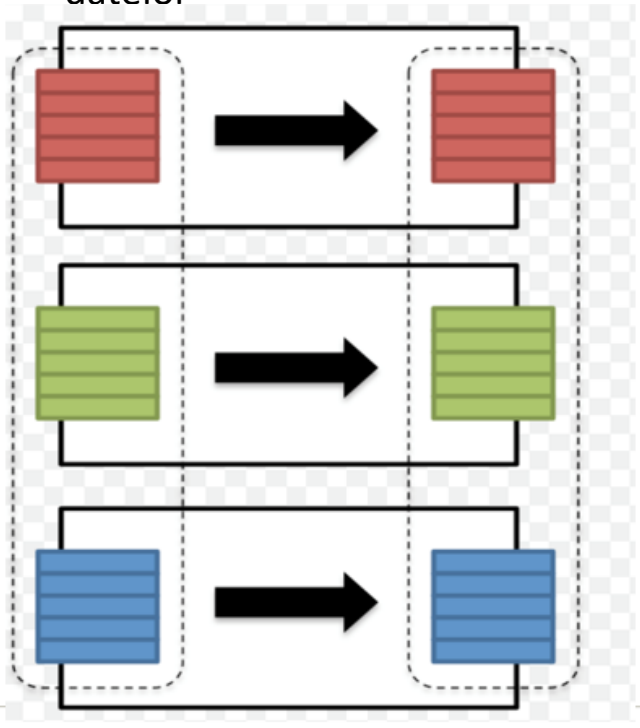


Dependența de tip *narrow*

- Relație 1:1 între partițiile copil-părinte
- Exemple de operații: ***Filter*** & ***Map***
- Proces relativ ieftin

Transformare narrow

- Intrarea și ieșirea rămân în aceeași partiție
- Nu este necesară mutarea datelor

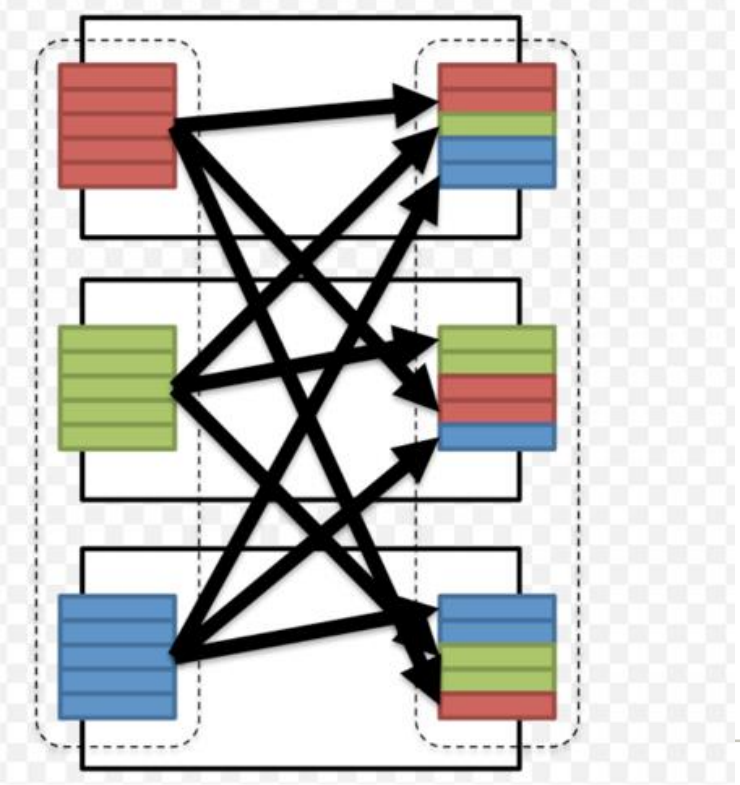


Dependența de tip *wide*

- Relație de tip M:1 sau M:M între partițiile copil-părinte
- Exemple de operații: ***Join*** & ***Grouping***
- Mai scumpe

Transformare wide

- Sunt necesare date de intrare din alte partiții
- Este necesară amestecarea datelor înainte de procesare



Metode asupra RDD-urilor

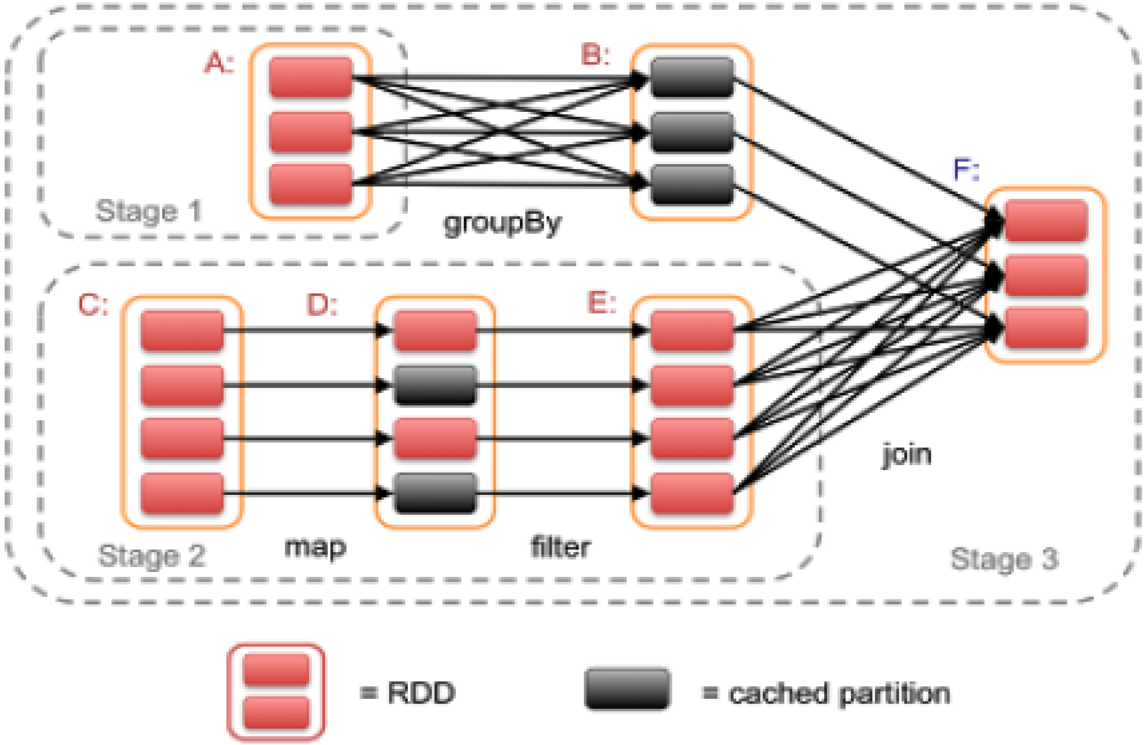
Operație	Semnificație
partitions()	Returnează o listă de obiecte de tip Partition
preferredLocations(p)	Listează nodurile în care partiția *p* poate fi accesată mai rapid datorită localizării datelor
dependencies()	Returnează o listă de dependențe
iterator(*p*, *parentIters*)	Calculează elementele partiției *p*; sunt dați iteratorii pentru partițiile sale părinte
partitioner()	Returnează metadatele ce specifică dacă RDD-ul este partiționat de tip hash/range

Metode (interfețe) utilizate pentru a reprezenta RDD-urile în Spark

Planificare și gestionare a memoriei

Planificare (*Scheduling*)

- Execuția este declanșată atunci când este invocată o operație de tip **acțiune**
- Planificatorul (*Scheduler*-ul) consultă graful de descendență pentru a putea executa



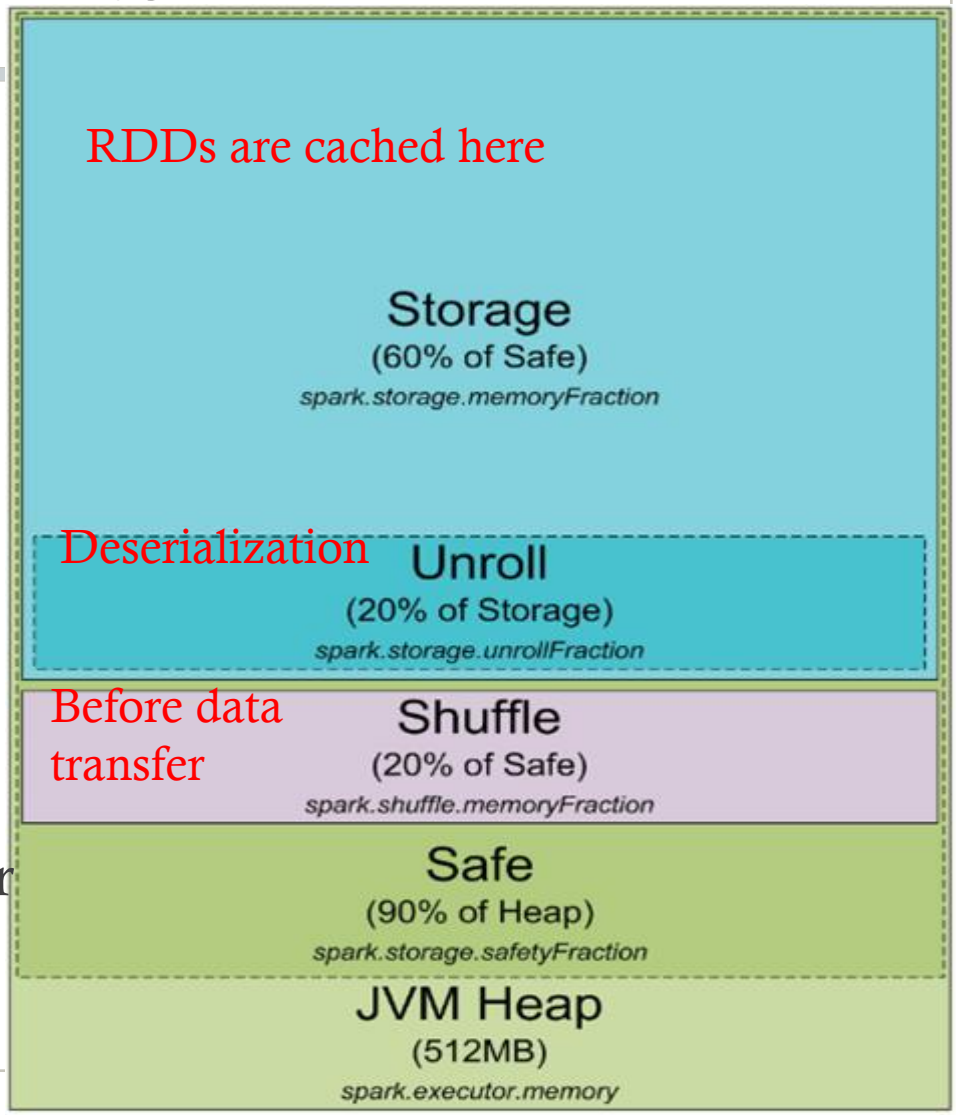
Gestionarea memoriei în Spark

- Utilizarea memoriei este esențială în Spark (Caching)
- Există 3 opțiuni pentru stocarea RDD-urilor persistente:

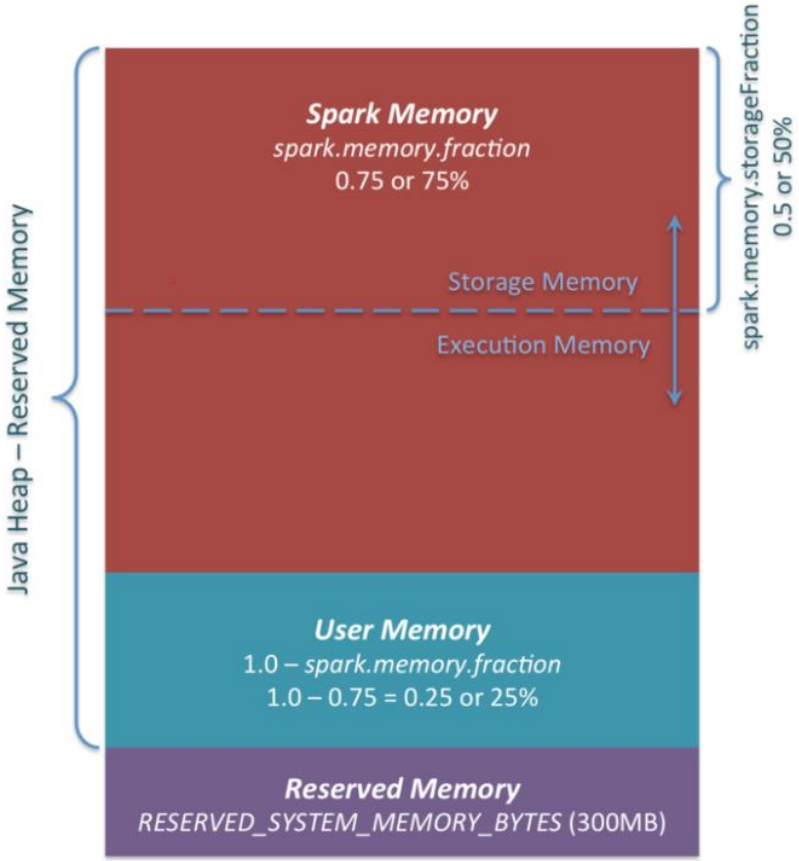
1. Stocare *in-memory* ca obiecte Java deserializate (performanța cea mai bună)
2. Stocare *in-memory* ca date serializate (eficient din punct de vedere al memoriei, dar cu performanțe mai reduse)
3. Stocare *on-disk* (RDD-ul este prea mare pentru a încăpea în memorie; costul cel mai ridicat)

Gestionarea memoriei în Spark <=1.5

- Procesul Spark este un proces JVM
- Memoria implicită este 512MB
- Există parametri pentru controlul utilizării segmentelor de memorie



Gestionarea memoriei în Spark 1.6.0+



Politica de înlocuire

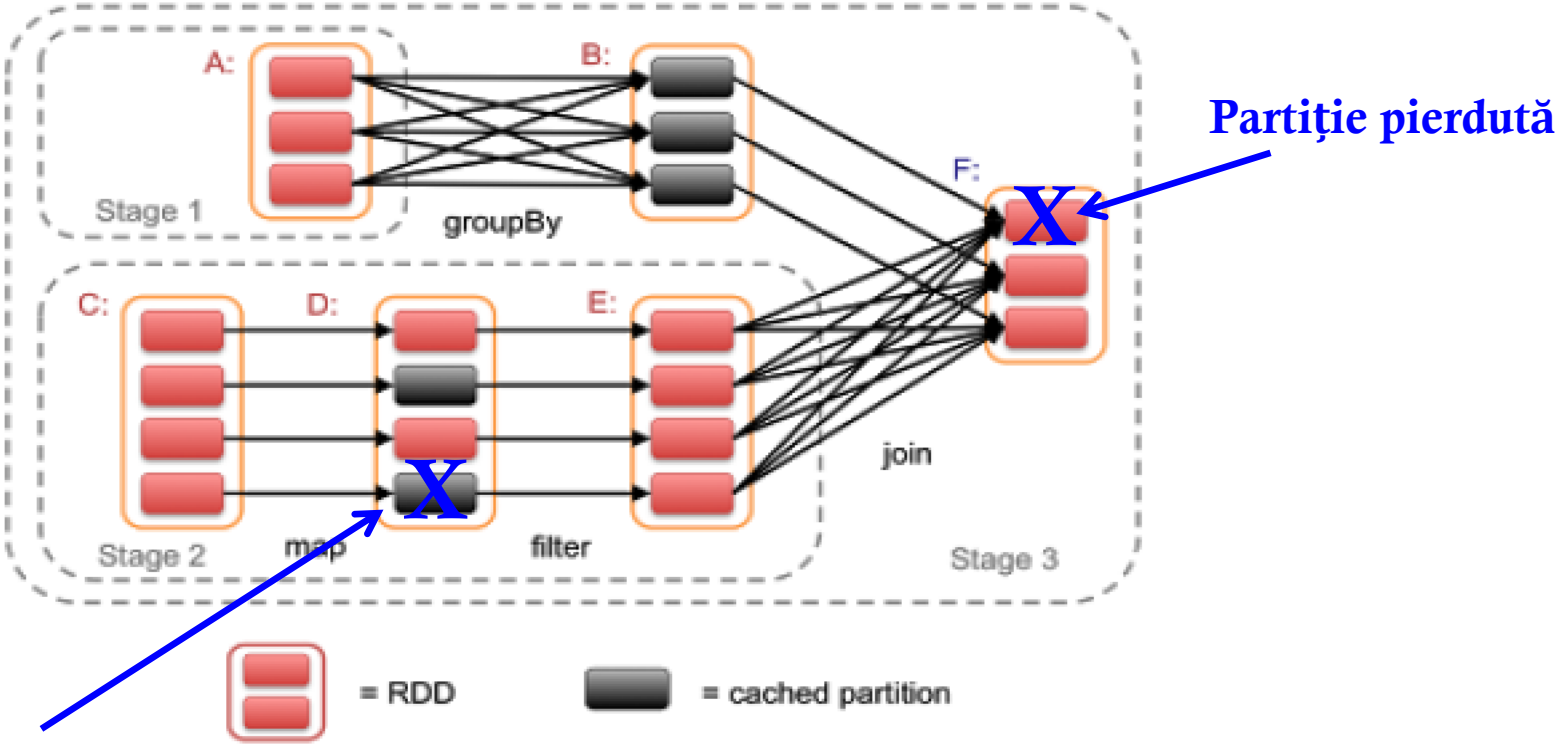
- Politica de eliminare folosită la nivelul partițiilor RDD este **LRU** (*Least Recently Used*)

- **Atunci când este creată o nouă partiție RDD:**
  - Dacă există spațiu în memorie ➔ Partiția se pune în *cache*
  - Dacă nu ➔ se elimină una sau mai multe partiții dintre cele LRU

- Se utilizează prioritatea persistenței ("*persistence priority*") pentru a preveni eliminarea unor RDD-uri importante

Recuperarea RDD-urilor

- În caz de eșec sau de pierdere a unei partiții RDD



Recuperarea RDD-urilor

- Recuperarea poate dura mult în cazul RDD-urilor cu lanțuri *lineage* lungi
- Spark oferă un API pentru *checkpointing* (un flag REPLICATE în metoda *persist*)
- Se poate utiliza mecanismul ***checkpointing*** pentru a determina ca unele RDD-uri să fie persistente, în mod:
  - Definit de utilizator sau
  - Controlat de sistem sau
  - Prin modalități mai specifice, de exemplu în funcție de volumul seturilor procesate

Concluzii (Spark vs Hadoop)

- Spark are performanțe mult mai bune -> 20x în ML iterativ și aplicațiile pe grafuri
- Viteza provine din evitarea costurilor cu operațiile I/O și ale deserializării, stocând datele în memorie ca obiecte Java
- Atunci când un nod eșuează, Spark poate recupera rapid reconstruind doar partițiile RDD pierdute
- Spark poate fi utilizat pentru interogarea unui dataset de 1TB în mod interactiv, cu latențe de 5-7 secunde.

Bibliografie

1. Matei Zaharia et al. “Spark: Cluster Computing with Working Sets”.

2. MateiZaharia et al. “Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing”.

3. Matei Zaharia, “An Architecture for Fast and General Data Processing on Large Clusters” (teza de doctorat).

4. Databricks resources (https://databricks.com/resources/slides).

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