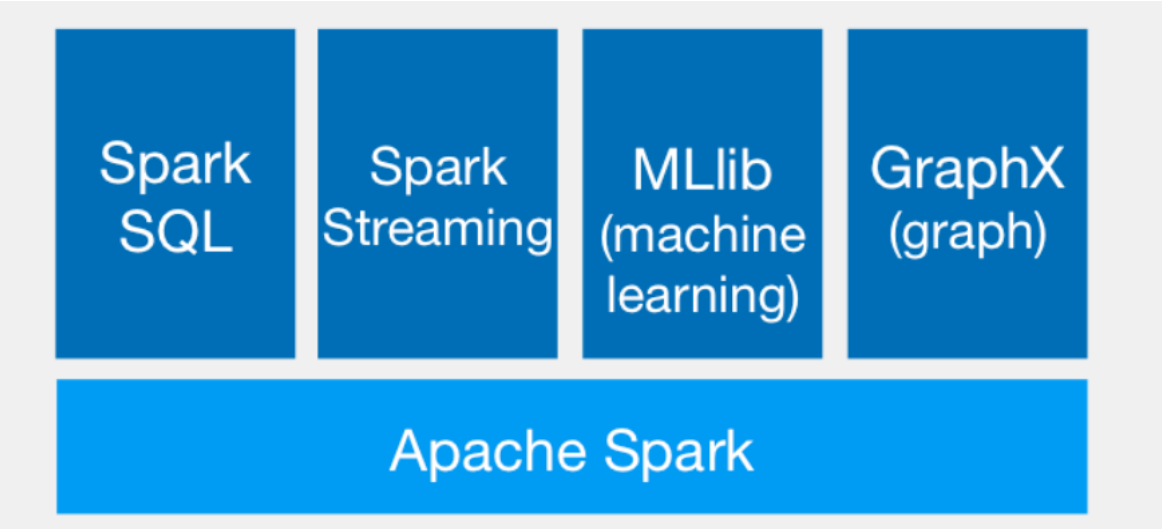
BIG DATA

CURS 4

Apache Spark SQL
(continuare)

Introducere

- Spark SQL este un modul Spark pentru procesarea datelor distribuite.
- Spark SQL este o componentă peste Spark Core, ce introduce un nou nivel de abstractizare.



Introducere

- Spark SQL a apărut în versiunea Spark 1.0 (Mai 2014).
- Inițial a fost dezvoltat de Michael Armbrust și Reynold Xin (Databricks).
- Spark introduce un modul pentru programare destinat procesării datelor structurate denumit Spark SQL.
- Acest modul furnizează o abstractizare denumită DataFrame și poate funcționa ca un motor de cereri SQL distribuite.
- Anterior versiunii Spark 1.3.0, DataFrame a fost denumit SchemaRDD.

Introducere

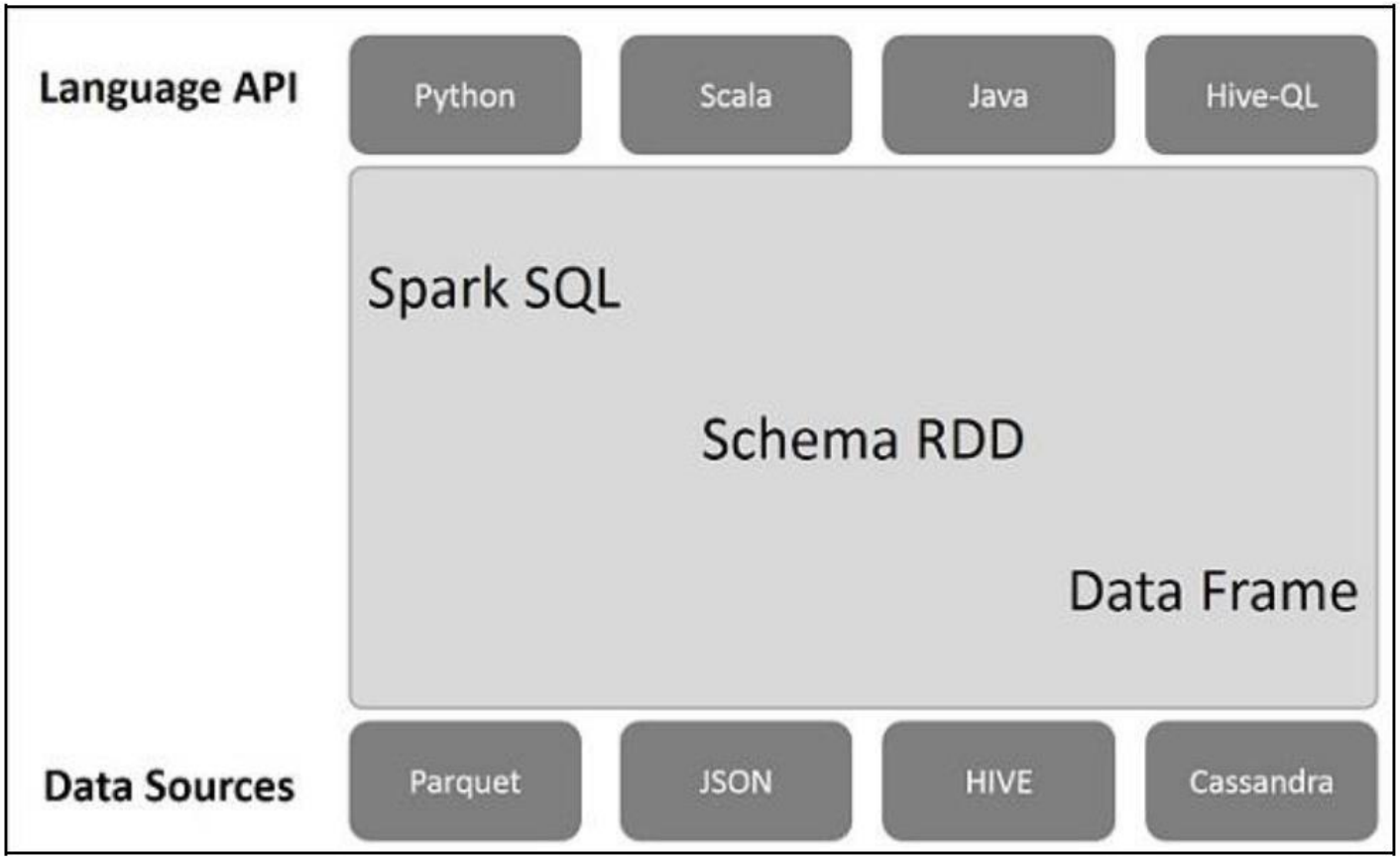
Probleme

- Realizarea ETL (Extract-Transform-Load) de la și către surse de date diferite (semistructurate sau nestructurate)
- Realizarea de analize avansate (de tip machine learning, procesare de grafuri) ce sunt dificil de exprimat în sistemele relaționale.

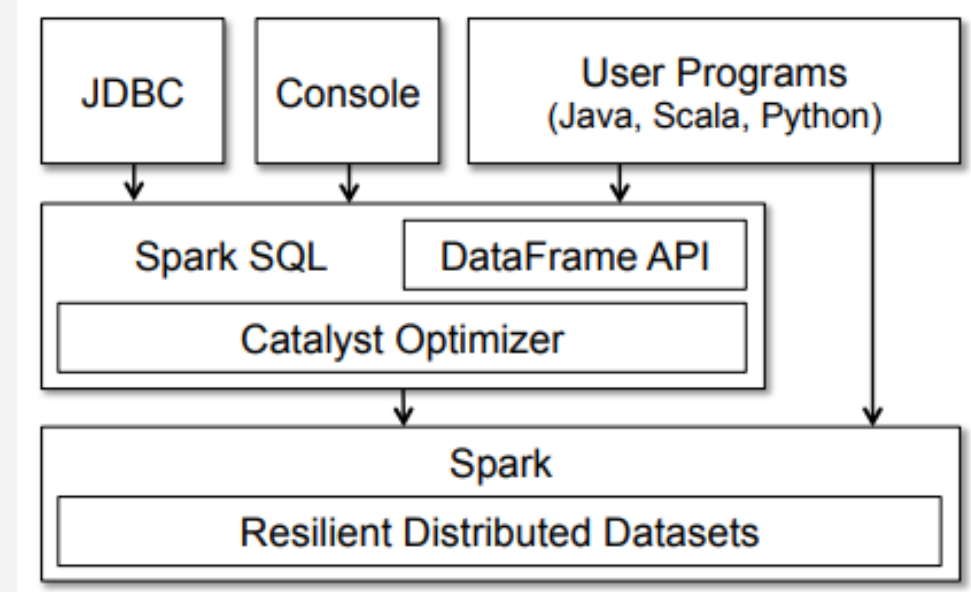
Soluții

- DataFrame API – ce poate realiza operații relaționale atât pe surse externe de date cât și pe structurile Spark predefinite (RDD).
- Un optimizor extensibil (Catalyst) ce utilizează caracteristicile limbajului Scala pentru a permite adăugarea de reguli ce pot fi compuse, controlul generării codului și definirea extensiilor.

Arhitectura Spark SQL



Arhitectura Spark SQL



Arhitectura Spark SQL

- Language API:
  - Spark este compatibil cu diferite limbaje (și Spark SQL)
  - Este suportat de către aceste limbaje prin intermediul API-urilor (Python, Java, Scala, HiveQL)
- SchemaRDD / DataFrame:
  - Structura de date centrală a lui Spark Core este RDD
  - În general, Spark SQL lucrează cu scheme, tabele și înregistrări.
  - Prin urmare, putem utiliza SchemaRDD ca tabel temporar.
  - Schema RDD -> Data Frame (în Spark >=1.3.0)
- Data Sources:
  - În general, sursa de date pentru Spark Core este un fișier text, Avro etc.
  - Sursele de date pentru Spark SQL sunt diferite: fișier Parquet, document JSON, tabele Hive, baze de date (Cassandra).

Caracteristici ale Spark SQL

1. Integrare:

- Combină cererile SQL cu programele Spark

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

Aplicare de funcții pe rezultatele cererilor SQL

- Spark SQL permite interogarea datelor structurate ca pe un RDD din Spark, cu API-uri integrate în Pyhon, Scala și Java.
- Această integrare facilitează execuția cererilor SQL în cadrul algoritmilor analitici complecși.

2. Acces unificat la date:

- Încarcă și interoghează date din surse diverse.

```
spark.read.json("s3n://...")
  .registerTempTable("json")
results = spark.sql(
  """SELECT *
     FROM people
     JOIN json ...""")
```

Interogări (cu join) pe surse multiple de date

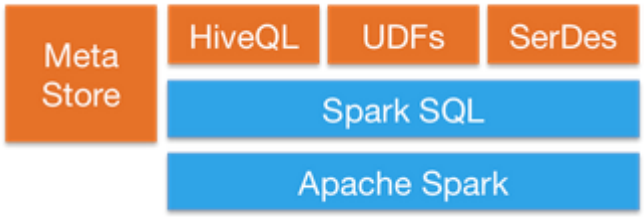
- DataFrame-urile furnizează o singură interfață pentru lucrul eficient cu date structurate, ce includ tabele Apache Hive, fișiere Parquet și fișiere JSON.

Caracteristici ale Spark SQL

3. Compatibilitatea cu Hive:

- Se pot executa cereri Hive nemodificate pe date warehouse-uri existente
- Spark SQL reutilizează frontend-ul Hive și Metastore, oferind compatibilitate deplină cu datele Hive, cererile și funcțiile definite de utilizator (UDF) existente.


```
SELECT
COUNT(*)
FROM
hiveTable
WHERE
hive_udf(data)
```

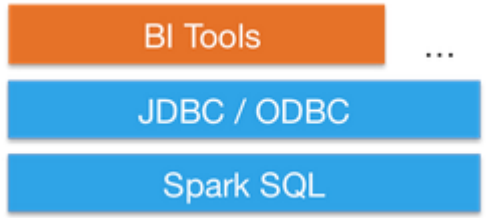


Spark SQL poate folosi metastore-uri, SerDes și UDF-uri din Hive

Caracteristici ale Spark SQL

4. Conectivitatea standard:

- Conexiuni prin JDBC sau ODBC
- Spark SQL include un mod server pentru tipurile de conectivitate standard din industrie (JDBC, ODBC) – pentru utilitarele de BI.



Se pot folosi tool-urile de BI existente pentru a interoga Big Data

Caracteristici ale Spark SQL

5. Performanță și scalabilitate:

- Include optimizor, stocare pe coloane și generare de cod pentru a îmbunătăți performanța cererilor
- Utilizarea aceluiași motor atât pentru cereri interactive, cât și pentru cele care necesită mult timp de procesare.
- Spark SQL beneficiază de avantajul modelului RDD pentru a suporta toleranța la defecte.
  - Scalează la mii de noduri și cereri de durată mare, folosind motorul Spark
- Poate fi utilizat un motor diferit pentru datele istorice.

SPARK SQL

DATASET AND DATAFRAME

Dataset și DataFrame

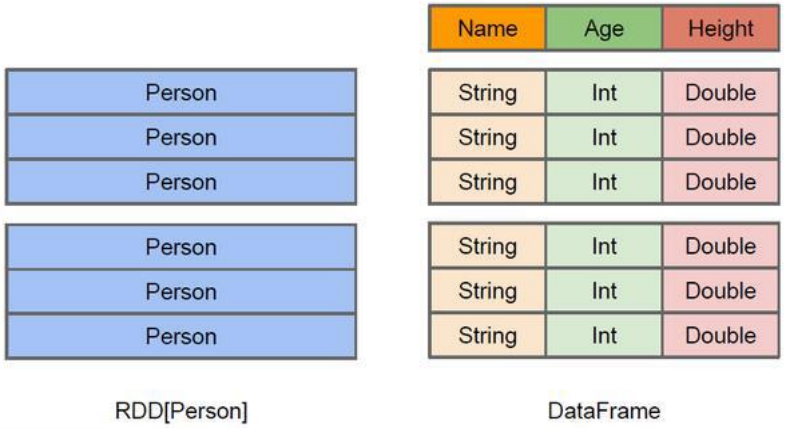
- Un dataset este o colecție distribuită de date.
- Este o interfață adăugată în versiunea 1.6.0 ce combină beneficiile RDD-urilor (*strong typing*, utilizarea de lambda funcții) cu beneficiile motorului de execuție optimizat din Spark SQL.
- Poate fi construit din obiecte JVM și apoi prelucrat cu ajutorul transformărilor funcționale (*map*, *flatMap*, *filter* etc.)
- Dataset API este disponibil pentru Java și Scala, nu și pentru Python.
  - Multe dintre funcționalitățile Dataset sunt disponibile și în Python, datorită naturii dinamice a acestuia.

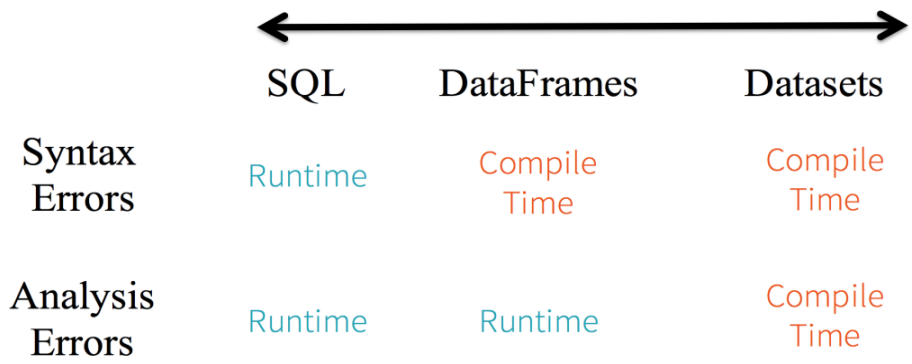
Dataset și DataFrame

- Un DataFrame este o colecție distribuită de date, organizată sub formă de coloane denumite.
  - Este un Dataset organizat sub formă de coloane denumite
- Conceptual, este echivalent cu tabelul relațional sau cu structurile de tip data frame din R și Python (din modulul Pandas), dar cu mai multe posibilități în privința optimizării.
- Un DataFrame poate fi construit din surse variate (tabele Hive, fișiere de date structurate, baze de date externe, RDD-uri existente).
- DataFrame API este disponibil în Java, Scala, Python și R.

Dataset și DataFrame

- DataFrame
  - Datele sunt organizate în coloane denumite, similar unui tabel dintr-o bază de date relațională
- Dataset: o colecție distribuită de date
  - O interfață adaugată în Spark 1.6
  - Permite static-typing și type-safety la runtime





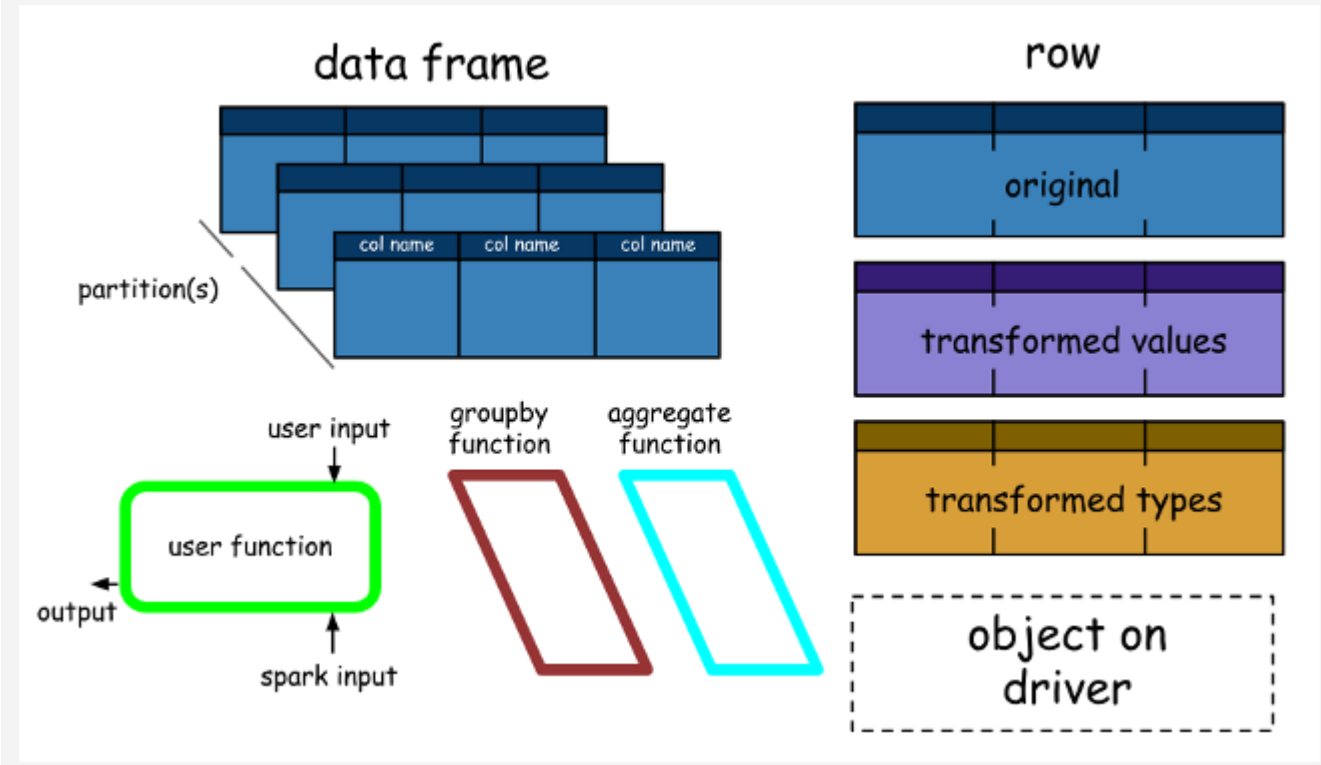
Caracteristicile unui DataFrame

- Capacitatea de a procesa date de ordin de la KB la Pentabytes pe clustere având de la un nod până la foarte multe noduri.
- Suportă diferite formate de date (Avro, csv, elastic search, Cassandra) și sisteme de stocare (HDFS, tabele Hive, tabele Mysql etc.)
- Optimizare și generare de cod cu ajutorul optimizorului Spark SQL Catalyst (transformare de arbori).
- Poate fi integrat cu ușurință cu toate utilitarele și framework-urile Big Data prin intermediul Spark Core.

Crearea unui DataFrame

- Punctul de intrare în funcționalitatea SQL din Spark este clasa **SQLContext**. Pentru a crea instanță de bază a acesteia, este necesar să referim un **SparkContext**.
- Vezi demo (final curs)

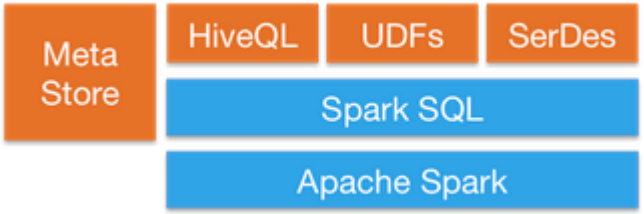
Crearea unui DataFrame



SPARK SQL și HIVE

Compatibilitatea cu Hive

- Spark SQL reutilizează frontend-ul și metastore-ul Hive, oferind compatibilitate deplină cu datele Hive, cererile și UDF-urile existente.



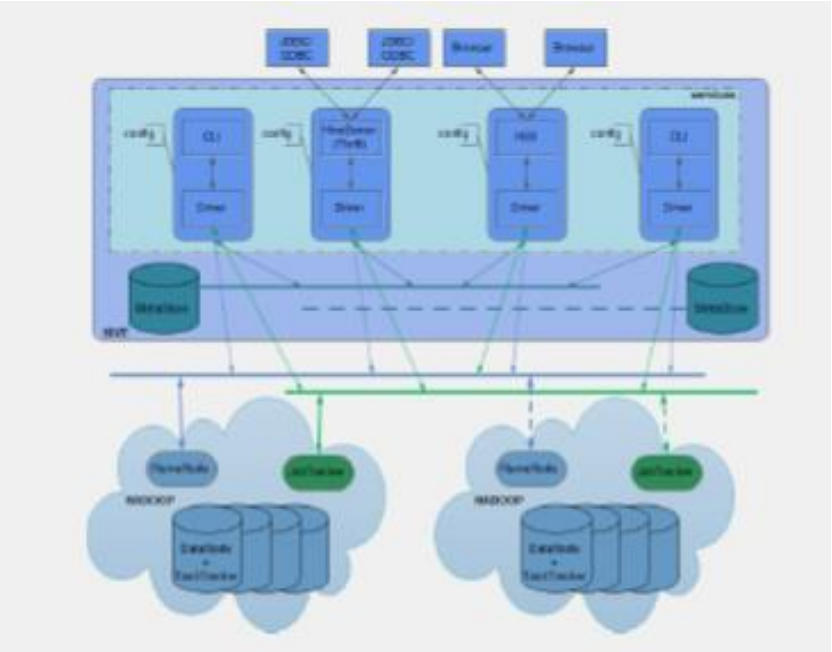
Spark SQL poate folosi metastore-uri, SerDes și UDF-uri din Hive

Hive

- O bază de date / data warehouse peste Hadoop
  - Tipuri de date numeroase
  - Implementări eficiente ale lui SQL peste map reduce
- Suportă analiza seturilor de date mari stocate în HDFS și sistemele de fișiere compatibile
  - Exemplu de sistem compatibil: Amazon S3
- Furnizează un limbaj asemănător lui SQL, denumit HiveQL.

Arhitectura Hive

- Utilizatorul lansează cererea SQL
- Hive parsează și determină planul de execuție
- Cererea este convertită în Map-Reduce
- Map-Reduce rulează prin Hadoop



Funcții definite de utilizator

- UDF: Dezvoltarea de cod sursă pentru procesarea specifică și invocarea lui dintr-o cerere Hive
  - UDF:
    - Intrarea: o înregistrare; Ieșirea : o înregistrare
  - UDAF (User Defined Aggregate Functions):
    - Intrarea: înregistrări multiple; Ieșirea: o înregistrare
    - De exemplu: COUNT și MAX
  - UDTF (User Defined Table-generating Functions)
    - Intrarea: o înregistrare; Ieșirea : mai multe întegistrări

SPARK SQL vs SQL

Spark SQL

- Spark SQL nu se referă doar la SQL
- Crearea și executarea mai rapidă a programelor:
  - Scriere de cod mai puțin
  - Citirea mai puținor date
  - Permite intervenția optimizorului

„Write less code” – I/O

- API-ul Spark SQL DataSource poate citi și scrie DataFrame-uri într-o varietate de formate

```
df = sqlContext.read \
  .format("json") \
  .option("samplingRatio", "0.1") \
  .load("/home/michael/data.json")

df.write \
  .format("parquet") \
  .mode("append") \
  .partitionBy("year") \
  .saveAsTable("fasterData")
```

Funcțiile *read* și *write* creează obiecte de tip builder pentru I/O

Metodele builder specifică: formatul, partiționarea, modul de tratare a datelor existente etc.

Funcțiile *load*, *save* și *saveAsTable* creează obiecte de tip builder pentru I/O

ETL folosind surse de date specifice

```
sqlContext.read
  .format("com.databricks.spark.git")
  .option("url",
  "https://github.com/apache/spark.git")
  .option("numPartitions", "100")
  .option("branches",
  "master,branch---1.3,branch---1.2")
  .load()
  .repartition(1)
  .write
  .format("json")
  .save("/home/michael/spark.json")
```

„Write less code” – Operații

- Operațiile obișnuite pot fi exprimate concis ca apeluri la API-ul DataFrame:
  - Selectarea anumitor coloane
  - Join între date din diferite surse
  - Agregări (count, sum, average etc.)
  - Filtrare

„Write less code” – calculul mediei


```
private IntWritable one =
  new IntWritable(1)
private IntWritable output =
  new IntWritable()
proctected void map(
    LongWritable key,
    Text value,
    Context context) {
  String[] fields = value.split("\t")
  output.set(Integer.parseInt(fields[1]))
  context.write(one, output)
}

IntWritable one = new IntWritable(1)
DoubleWritable average = new DoubleWritable()

protected void reduce(
    IntWritable key,
    Iterable<IntWritable> values,
    Context context) {
  int sum = 0
  int count = 0
  for(IntWritable value : values) {
     sum += value.get()
     count++
    }
  average.set(sum / (double) count)
  context.Write(key, average)
}
```


```
data = sc.textFile(...).split("\t")
data.map(lambda x: (x[0], [x.[1], 1])) \
   .reduceByKey(lambda x, y: [x[0] + y[0], x[1] + y[1]]) \
   .map(lambda x: [x[0], x[1][0] / x[1][1]]) \
   .collect()
```

„Write less code” – calculul mediei

Cu ajutorul RDD-urilor:

```
data = sc.textFile(...).split("\t")
data.map(lambda x: (x[0], [int(x[1]), 1])) \
    .reduceByKey(lambda x, y: [x[0] + y[0], x[1] + y[1]]) \
    .map(lambda x: [x[0], x[1][0] / x[1][1]]) \
    .collect()
```

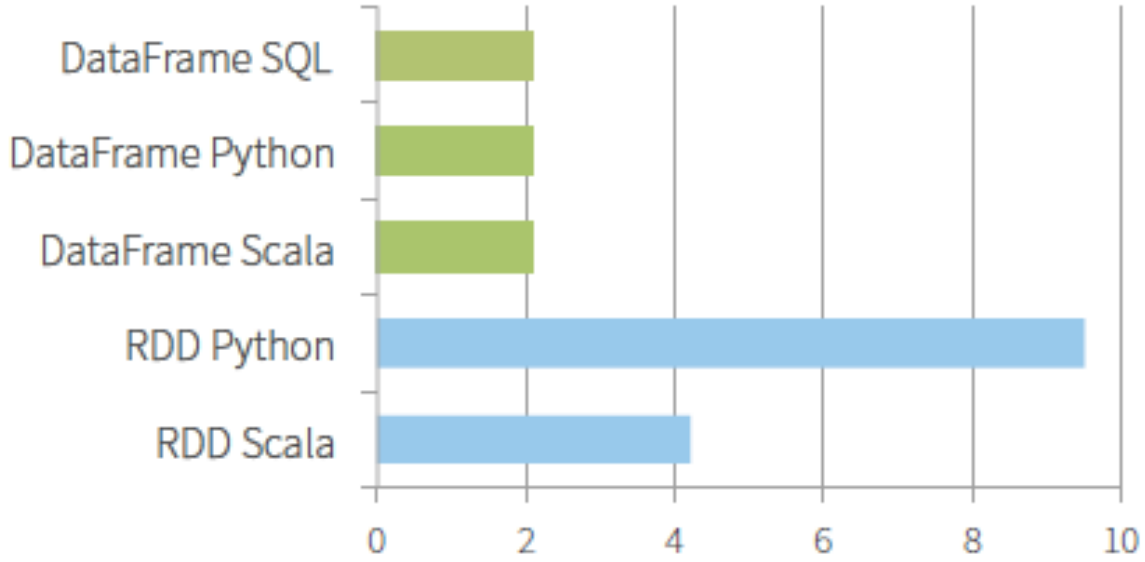
Cu ajutorul SQL:

```
SELECT name, avg(age)
FROM people
GROUP BY name
```

Cu ajutorul DataFrame-urilor:

```
sqlCtx.table("people") \
    .groupBy("name") \
    .agg("name", avg("age")) \
    .collect()
```

Implementări mai rapide



Timpul pentru agregarea a 10 milioane de perechi de valori int (în secunde)

„Read less data”

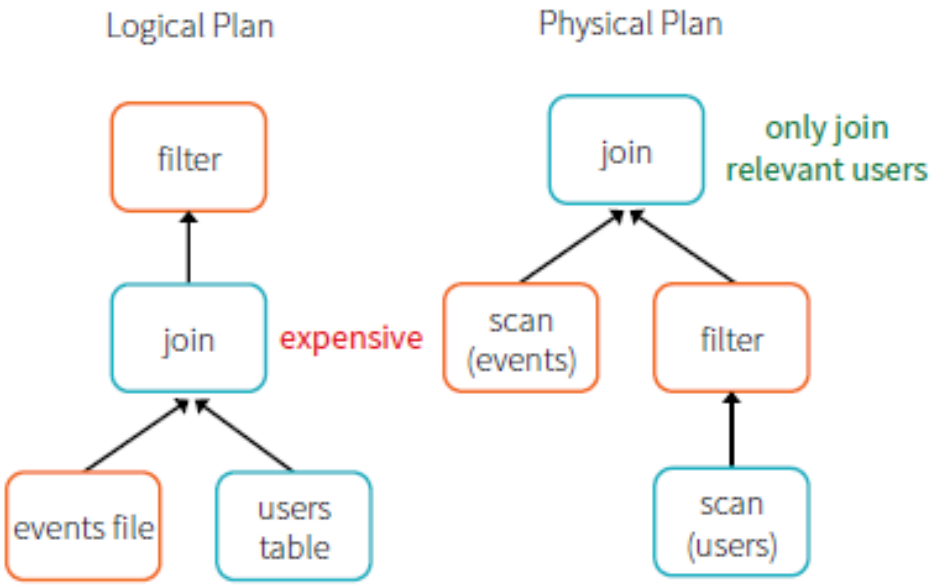
- Spark SQL permite automat citirea mai puținor date:
  - Convertind în formate mai eficiente
  - Folosind formatele pe coloane (de exemplu, Parquet)
  - Folosind partiționările
  - Folosind statistici asupra datelor în loc de date efective (de exemplu, min, max etc.)
  - Permutarea unor predicate

Optimizare

- Optimizarea are loc după etapa de analiză
- Optimizorul Spark (Catalyst) suportă optimizarea bazată pe reguli și cea bazată pe cost
- În optimizarea bazată pe reguli (RBO), definim un set de reguli care vor determina modul în care va fi executată cererea. Aceste reguli vor rescrie cererea astfel încât performanța să fie îmbunătățită.
- RBO nu ia în considerare distribuirea datelor. Aici ne putem îndrepta atenția spre optimizarea bazată pe cost (CBO). Aceasta utilizează statistici despre tabel, indecșii săi și distribuirea datelor.

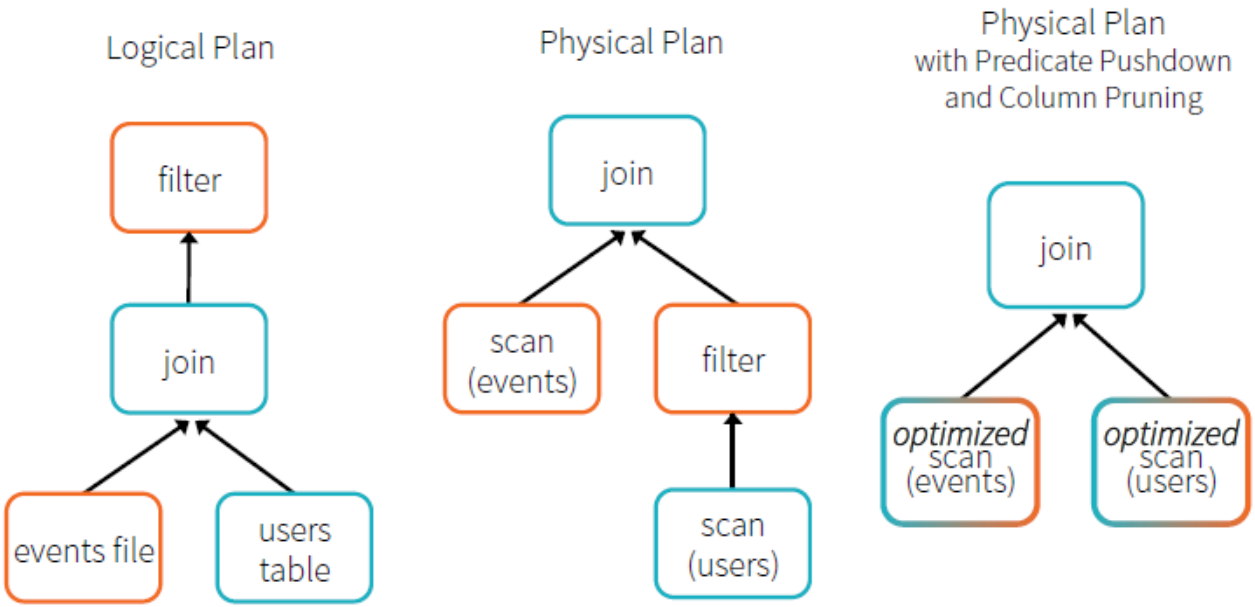
Optimizare

```
def add_demographics(events):
   u = sqlCtx.table("users")                    # Load Hive table
   events \
     .join(u, events.user_id == u.user_id) \    # Join on user_id
     .withColumn("city", zipToCity(df.zip))     # udf adds city column

events = add_demographics(sqlCtx.load("/data/events", "json"))
training_data = events.where(events.city == "Palo Alto")
                      .select(events.timestamp).collect()
```



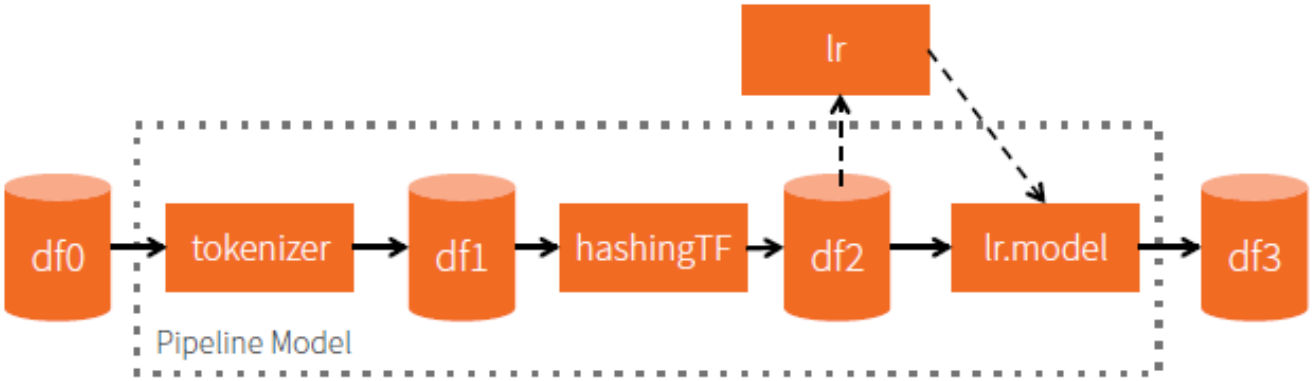
Optimizare

```
def add_demographics(events):
    u = sqlCtx.table("users")                    # Load partitioned Hive table
    events \
      .join(u, events.user_id == u.user_id) \    # Join on user_id
      .withColumn("city", zipToCity(df.zip))     # Run udf to add city column

events = add_demographics(sqlCtx.load("/data/events", "parquet"))
training_data = events.where(events.city == "Palo Alto")
                      .select(events.timestamp).collect()
```



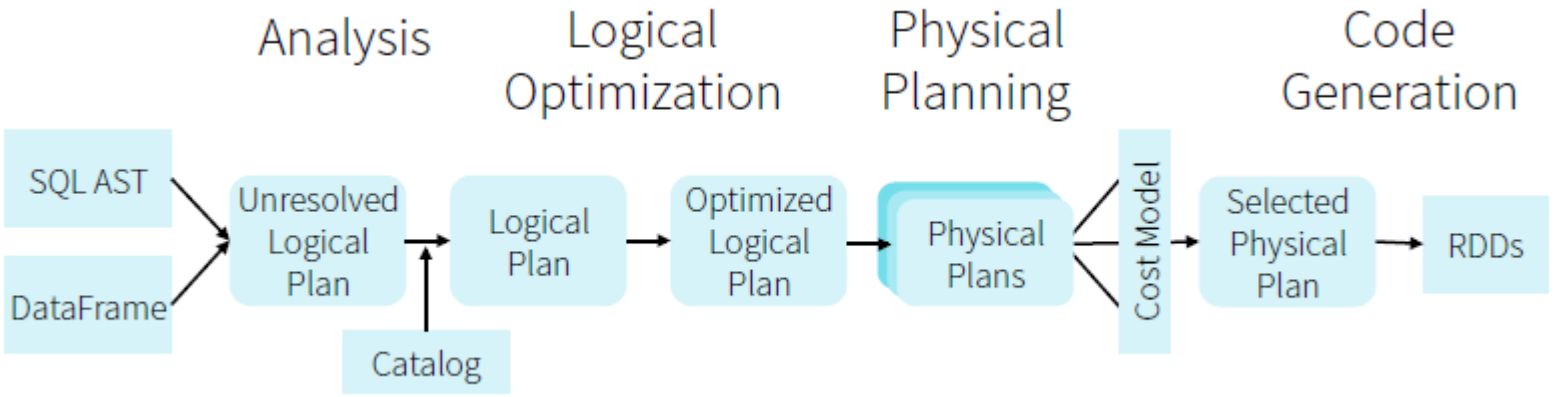
Pipeline ML

```
tokenizer = Tokenizer(inputCol="text", outputCol="words”)
hashingTF = HashingTF(inputCol="words", outputCol="features”)
lr = LogisticRegression(maxIter=10, regParam=0.01)
pipeline = Pipeline(stages=[tokenizer, hashingTF, lr])

df = sqlCtx.load("/path/to/data")
model = pipeline.fit(df)
```



Cum funcționează?

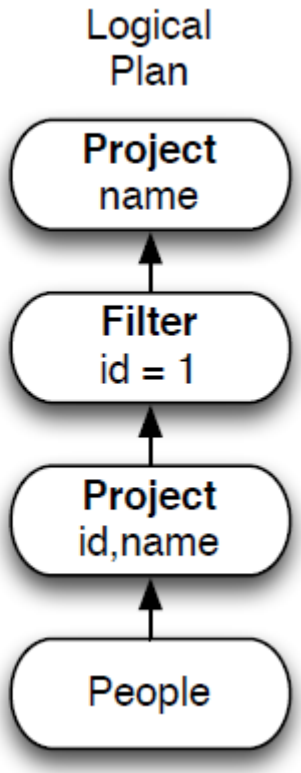
- Plan de optimizare și execuție



- DataFrame-urile și SQL partajează același *pipeline* pentru optimizare/execuție

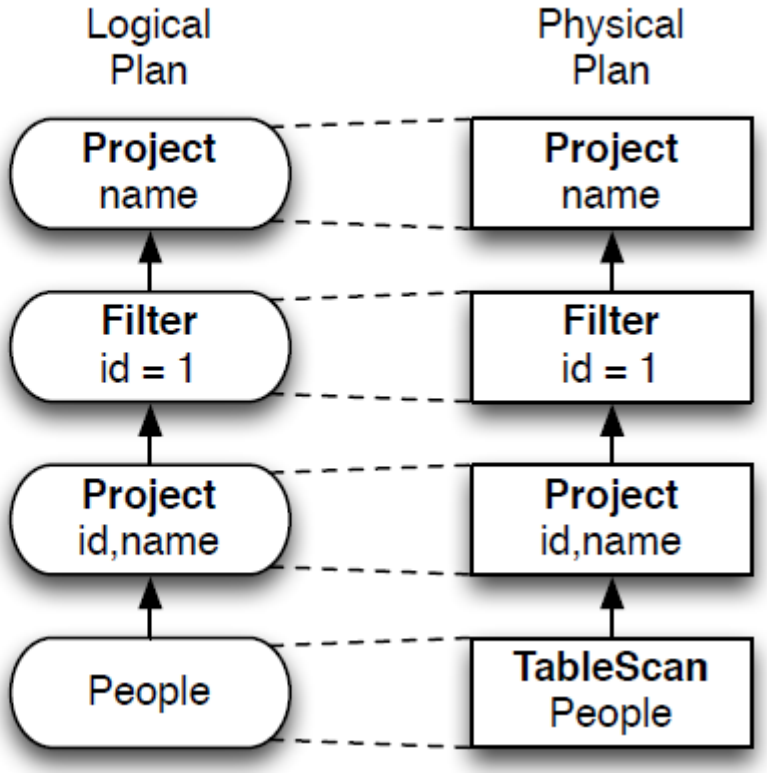
Exemplu

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

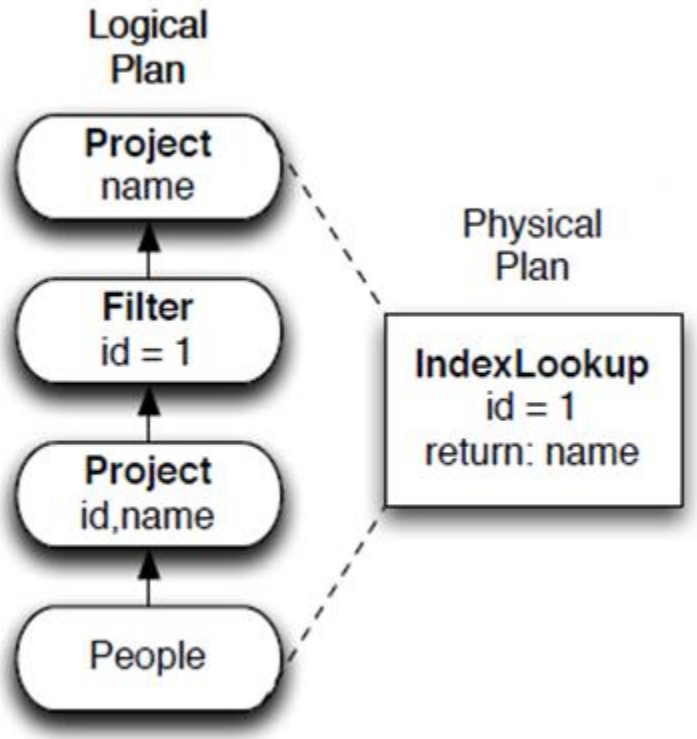


Planificarea cererii

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



Execuție optimizată



Scrierea de cod imperativ pentru a optimiza toate pattern-urile posibile ar fi dificilă.

Se scriu reguli simple:

- Fiecare regulă face o singură schimbare
- Se rulează mai multe reguli

Librăria TreeNode

- Arbori de operatori ce pot fi transformați ușor
  - Funcționalități standard ale colecțiilor: *foreach*, *map*, *collect* etc.
  - Funcție *transform* – modificarea recursivă a fragmentelor de arbore ce corespund unui pattern
  - Suport pentru debugging – *pretty printing*, divizare etc.

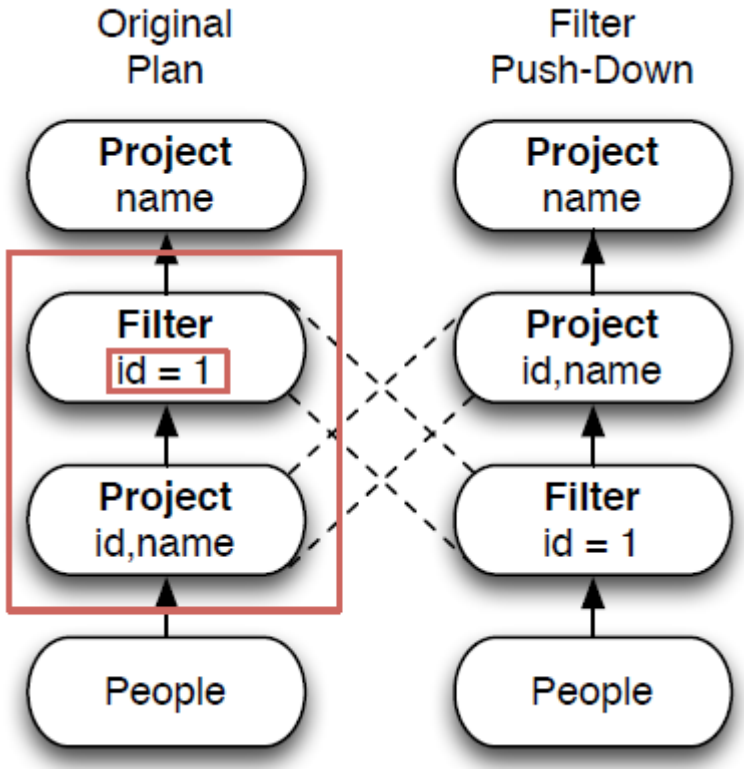
Transformări pe arbori

- Dezvoltatorii exprimă transformările pe arbori ca `PartialFunction[TreeType, TreeType]`

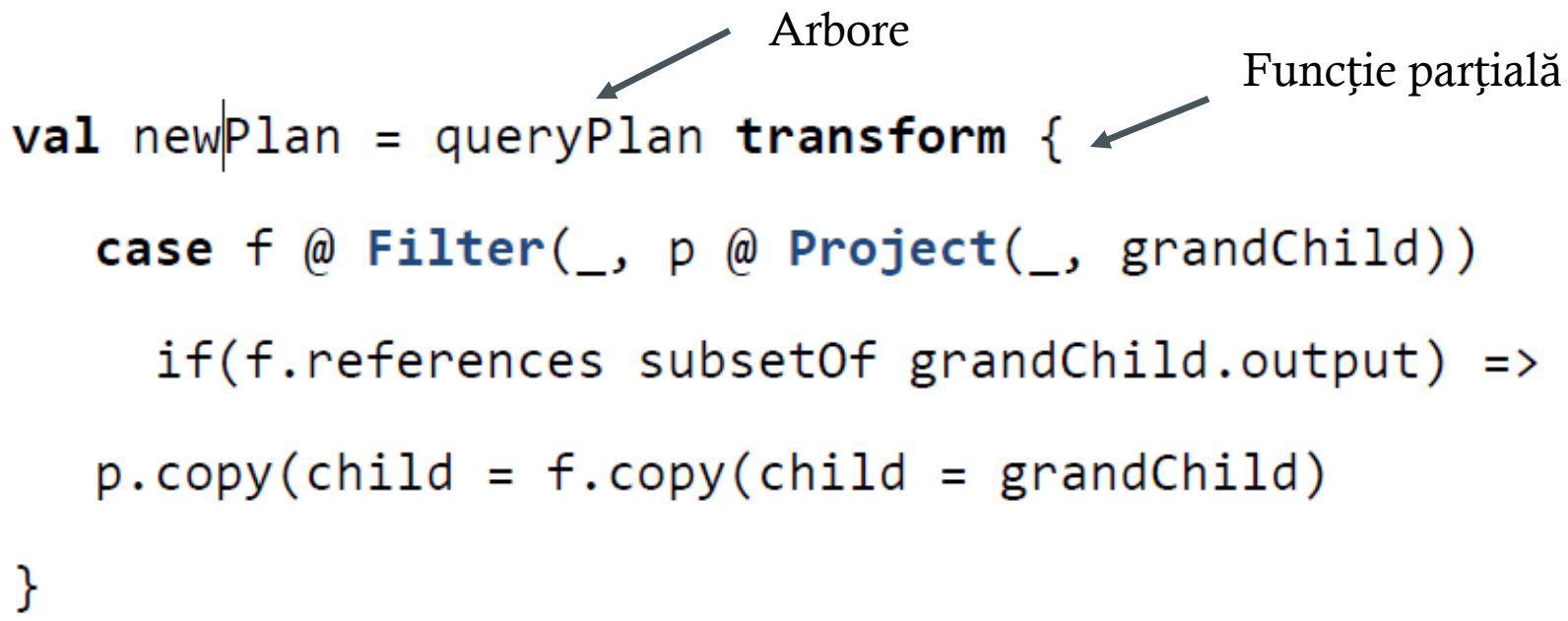
1. Dacă funcția se aplică unui operator, acel operator va fi înlocuit cu rezultatul funcției.
2. Atunci când funcția nu se aplică unui operator, acel operator rămâne neschimbat.
3. Transformarea este aplicată recursiv pe toate nodurile copil.

Scrierea de reguli ca operații de transformare pe arbori

1. Se determină filtrările care au loc deasupra proiecțiilor.
2. Se verifică dacă filtrul poate fi evaluat fără a fi necesar rezultatul proiecției.
3. În caz afirmativ, cei 2 operatori se interschimbă.



Coborârea operațiilor de filtrare



Coborârea operațiilor de filtrare

Se determină filtrarea asupra proiecției

```
val newPlan = queryPlan transform {
  case f @ Filter(_, p @ Project(_, grandChild))
    if(f.references subsetOf grandChild.output) =>
  p.copy(child = f.copy(child = grandChild)
}
```

Coborârea operațiilor de filtrare

```
val newPlan = queryPlan transform {
   case f @ Filter(_, p @ Project(_, grandChild))
     if(f.references subsetOf grandChild.output) =>
   p.copy(child = f.copy(child = grandChild)
}
```

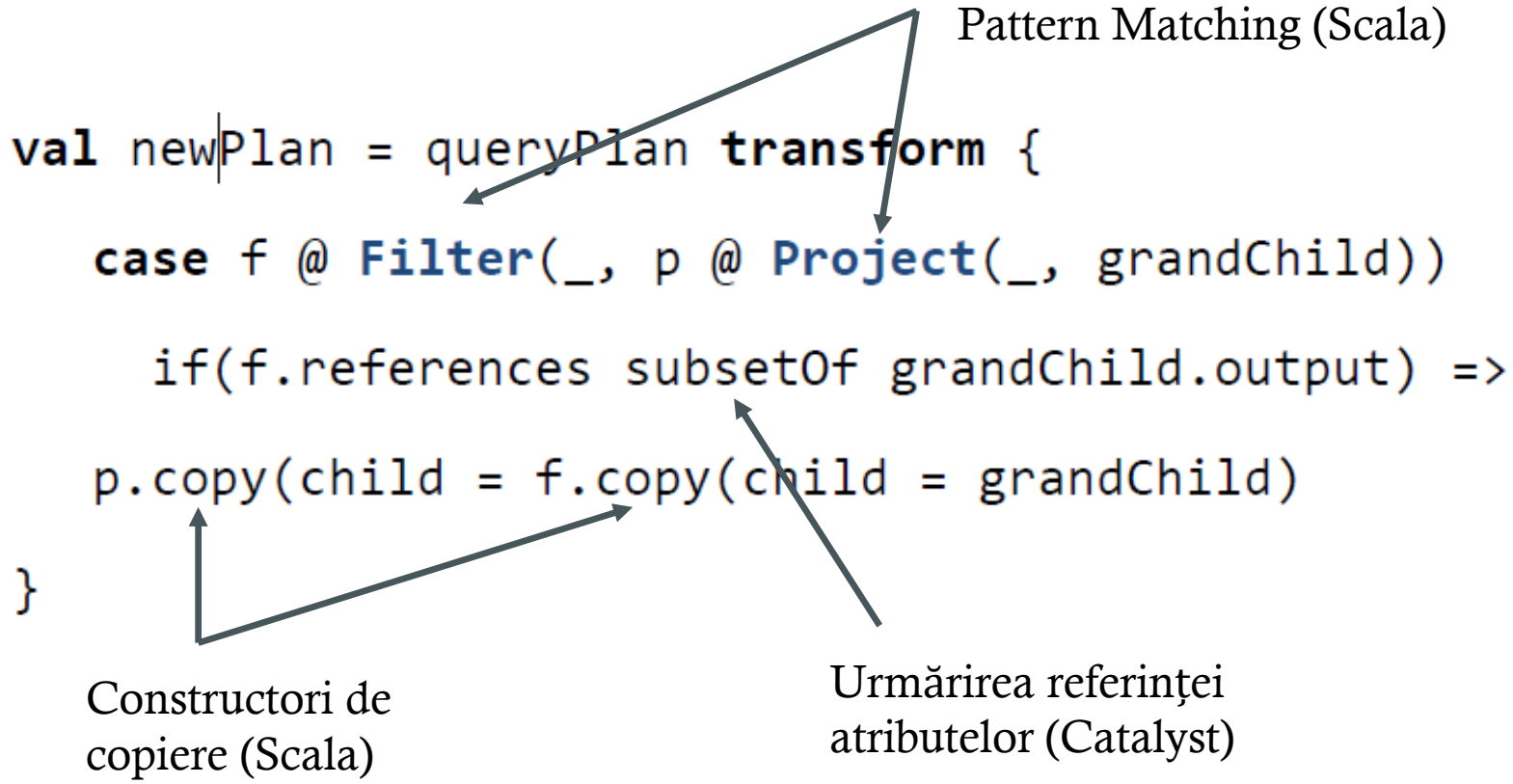
Se verifică dacă filtrul poate fi aplicat fără rezultatul proiecției

Coborârea operațiilor de filtrare

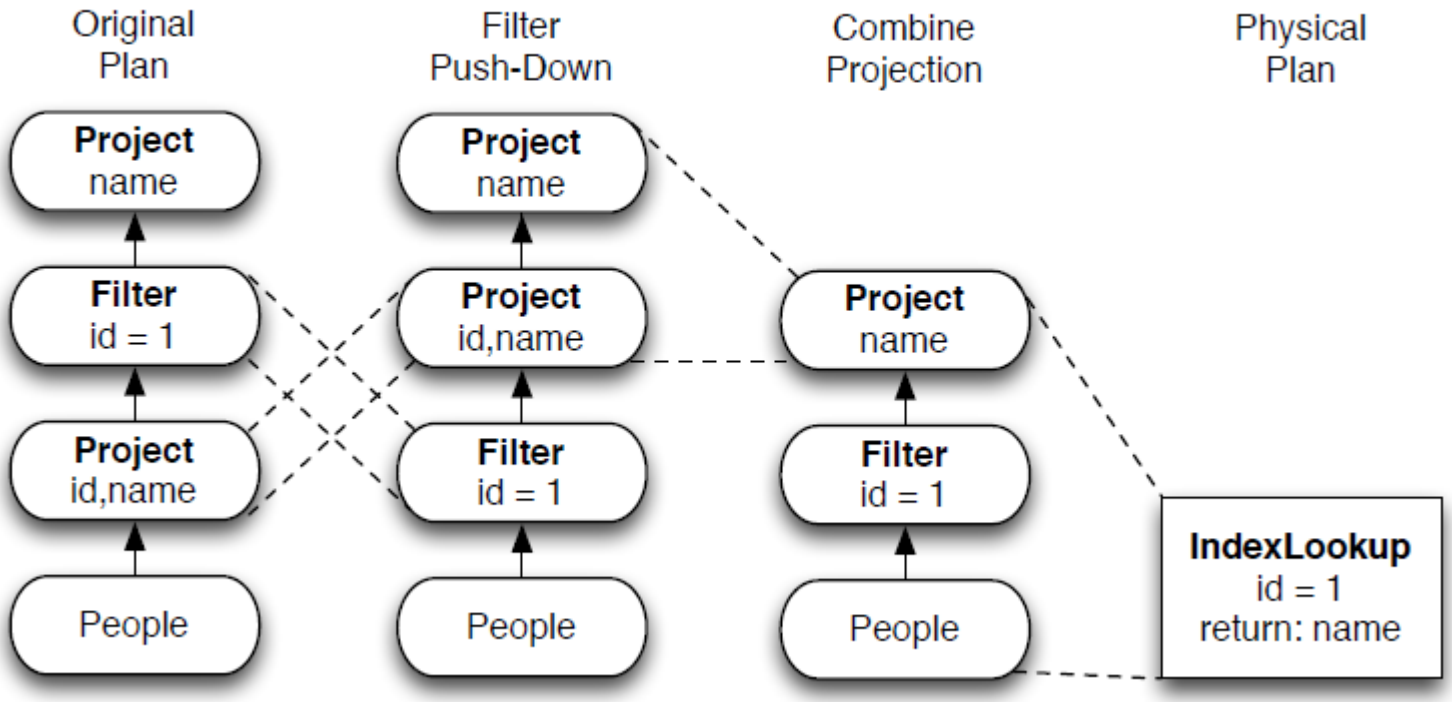
```
val newPlan = queryPlan transform {
   case f @ Filter(_, p @ Project(_, grandChild))
     if(f.references subsetOf grandChild.output) =>
   p.copy(child = f.copy(child = grandChild)
}
```

În caz afirmativ, se schimbă ordinea operațiilor

Coborârea operațiilor de filtrare



Optimizarea cu reguli



Demo 1

- Se va folosi setul de date https://drive.google.com/file/d/1sy5G4Y1RnlndLK86jzcV3CZJcTeo8Foj
- Setul conţine 25 milioane linii.
- Se va importa setul, se va citi şi se va diviza în partiţii
- Afişaţi primele 2 linii şi determinaţi structura lor. Se vor observa valori reprezentând: vârsta, grupa sanguină, oraşul, genul, id-ul).
- Rândurile vor fi mapate în obiecte de tip Row, având tipurile de date corespunzătoare pentru valori (atribute)
- Se va crea DataFrame-ul corespunzător setului de date
- Se cere numărul de înregistrări pentru fiecare gen, în două moduri (pe dataframe şi cu tabel temporar). Comparaţi timpul de execuţie al celor 2 metode.
- Calculaţi vârsta medie a persoanelor din fiecare oraş.

Demo 2

- Se va folosi setul de date kddcup.data_10_percent.gz de la adresa http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html
- Setul redus (10%) conţine aproximativ 500 000 interacţiuni în reţea.
- Se va încărca setul într-un RDD
- Divizaţi datele CSV şi afişaţi primele 2 valori
- Determinaţi numărul de caracteristici (coloane)
- Lista numelor acestor coloane se află în fişierul kddcup.names , iar descrierea lor pe pagina http://kdd.ics.uci.edu/databases/kddcup99/task.html
- Coloanele pe care le vom utiliza sunt: duration, protocol_type, service, src_bytes, dst_bytes, flag, wrong_fragment, urgent, hot,num_failed_logins, num_compromised, su_attempted, num_root, num_file_creations

Demo 2 (continuare)

- Se vor extrage coloanele precedente într-un alt RDD
- Se va crea DateFrame-ul corespunzător
- Se vor afișa primele 10 rânduri și schema datelor
- Se va crea un tabel temporar corespunzător DataFrame-ului
- Să se determine numărul de conexiuni pentru fiecare tip de protocol (prin ambele metode).
- Să se determine numărul de conexiuni pentru fiecare tip de semnătură (label).
- Utilizati Pandas Dataframe și matplotlib pentru a afișa grafic rezultatul anterior (bar chart).
- Determinați numărul de conexiuni pe baza protocoalelor și a atacurilor.
- Afișați statistici asupra conexiunilor de tip tcp, pentru fiecare tip de serviciu și de atac.

Bibliografie

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

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

3. Apache Spark SQL Programming Guide (http://spark.apache.org/docs/latest/sql-programming-guide.html)

4. http://parquet.incubator.apache.org/

5. https://towardsdatascience.com/pyspark-and-sparksql-basics-6cb4bf967e53