



Secondo giorno: mattina

# Agenda 3/8

## — API

### DataFrames e DataSet

In profondità

### Gli schemi

Nome e tipo delle colonne; definiti manualmente o letti da un *data source*.

### Colonne ed espressioni

Vari tipi di trasformazioni su DataFrame e DataSet; selezioni, filtri, ordinamenti

### I/O

Lettura e scrittura di file; HDFS, Hive, Local FS, S3, SequenceFile,...

### Gestire le stringhe

Trimming, Split (Explode), Espressioni regolari, Date e TimeStamp, Json

### Funzioni definite dall'utente

Le UDF

### SQL

### Esercizi di laboratorio

Esercitazioni sugli argomenti trattati.

```
root
|-- InvoiceNo: string (nullable = true)
|-- StockCode: string (nullable = true)
|-- Description: string (nullable = true)
|-- Quantity: integer (nullable = true)
|-- InvoiceDate: timestamp (nullable = true)
|-- UnitPrice: double (nullable = true)
|-- CustomerID: double (nullable = true)
|-- Country: string (nullable = true)
```

InvoiceNo	StockCode	Description	Quantity	InvoiceDate	Unit...
536365	85123A	WHITE HANGING HEA...	6	2010-12-01 08:26:00	...
536365	71053	WHITE METAL LANTERN	6	2010-12-01 08:26:00	...
...					
536367	21755	LOVE BUILDING BLO...	3	2010-12-01 08:34:00	...
536367	21777	RECIPE BOX WITH M...	4	2010-12-01 08:34:00	...

# DataFrame e Dataset

La principale API di Spark (nel 2019!)

```
val lst = spark.read
  .option("header",true)
  .option("inferSchema",true)
  .option("mode","FAILFAST")
  .csv("data/earthquake.csv")
```

lst: org.apache.spark.sql.DataFrame = [id: decimal(3,-11), date: string ... 15 more fields]

Took: 7.689s, at 2019-02-28 10:52

```
lst.show()
```

id	date	time	lat	long	country	city	area	direction	dist	depth	xm	md	richter	mw	ms	mb
2.00E+13	2003.05.20	12:17:44 AM	39.04	40.38	turkey	bingol	baliklicay	west	0.1	10.0	4.1	4.1	0.0	null	0.0	0.0
2.01E+13	2007.08.01	12:03:08 AM	40.79	30.09	turkey	kocaeli	bayraktar_izmit	west	0.1	5.2	4.0	3.8	4.0	null	0.0	0.0
1.98E+13	1978.05.07	12:41:37 AM	38.58	27.61	turkey	manisa	hamzabeyli	south_west	0.1	0.0	3.7	0.0	0.0	null	0.0	3.7
2.00E+13	1997.03.22	12:31:45 AM	39.47	36.44	turkey	sivas	kahvepinar_sarkisla	south_west	0.1	10.0	3.5	3.5	0.0	null	0.0	0.0
2.00E+13	2000.04.02	12:57:38 AM	40.8	30.24	turkey	sakarya	meseli_serdivan	south_west	0.1	7.0	4.3	4.3	0.0	null	0.0	0.0
2.01E+13	2005.01.21	12:04:03 AM	37.11	27.75	turkey	mugla	demirciler_milas	south_west	0.1	32.8	3.5	3.5	0.0	null	0.0	0.0
2.01E+13	2012.06.24	12:07:22 AM	38.75	43.61	turkey	van	ilikaynak	south_west	0.1	9.4	4.5	0.0	4.5	null	0.0	0.0
1.99E+13	1987.12.31	12:49:54 AM	39.43	27.98	turkey	balikesir	dikkonak_bigadic	south_east	0.1	26.0	3.8	3.8	0.0	null	0.0	0.0
2.00E+13	2000.02.07	12:11:45 AM	40.05	34.07	turkey	kirikkale	kocabas_delice	south_east	0.1	1.0	3.8	3.8	0.0	null	0.0	0.0
2.01E+13	2011.10.28	12:47:56 AM	38.76	43.54	turkey	van	degirmenozu	south_east	0.1	3.1	4.3	0.0	4.2	null	0.0	4.3
2.01E+13	2013.05.01	12:47:56 AM	37.31	37.11	turkey	kahramanmaras	ordekdede_pazarcik	south_east	0.1	9.5	3.5	0.0	3.5	null	0.0	0.0
1.99E+13	1989.04.27	12:45:19 AM	37.04	28.04	turkey	mugla	kultak_milas	south	0.1	9.0	3.6	3.6	0.0	null	0.0	0.0
2.00E+13	1999.11.26	12:42:20 AM	37.77	38.54	turkey	adiyaman	zeytin_kahta	south	0.1	13.0	3.6	3.6	0.0	null	0.0	0.0
2.00E+13	1999.12.20	12:41:56 AM	40.86	30.99	turkey	duzce	adakoy_gumusova	south	0.1	9.0	3.6	3.6	0.0	null	0.0	0.0
1.98E+13	1984.02.02	12:10:29 AM	37.21	30.81	turkey	antalya	kayadibi_aksu	north_west	0.1	15.0	3.7	0.0	0.0	null	0.0	3.7
2.01E+13	2011.05.22	12:49:49 AM	39.13	29.04	turkey	kutahya	kapikaya_simav	north_west	0.1	7.2	3.9	0.0	3.9	null	0.0	0.0
1.97E+13	1971.05.20	12:08:46 AM	37.72	30.0	turkey	burdur	kavacik	north_east	0.1	5.0	3.5	3.5	0.0	null	0.0	0.0
1.99E+13	1985.01.28	12:20:56 AM	38.85	29.06	turkey	manisa	karakozan_selendi	north_east	0.1	4.0	3.7	0.0	0.0	null	0.0	3.7
2.00E+13	1997.05.31	12:59:03 AM	39.89	39.79	turkey	erzincan	baskoy_cayirli	north_east	0.1	26.0	3.5	3.5	0.0	null	0.0	0.0
2.01E+13	2005.07.24	12:36:10 AM	36.96	36.03	turkey	hatay	turuncu_erzin	north_east	0.1	22.0	4.1	0.0	4.1	null	0.0	0.0

only showing top 20 rows

Took: 1.411s, at 2019-02-28 10:54

# DataFrame

- La più comune API strutturata di Spark
- Simile ad uno spreadsheet o alla tabella di un DB
- Deriva (in parte) da HiveQL
- Viene considerata una API «untyped»: Il tipo viene controllato solo a run-time
- Dopo Spark 2.0: `DataFrame = Dataset [Row]`
- La classe *Row* è una rappresentazione interna di Spark, ottimizzata per il processing
- I DataFrame sono disponibili in tutti i linguaggi

# DataFrame

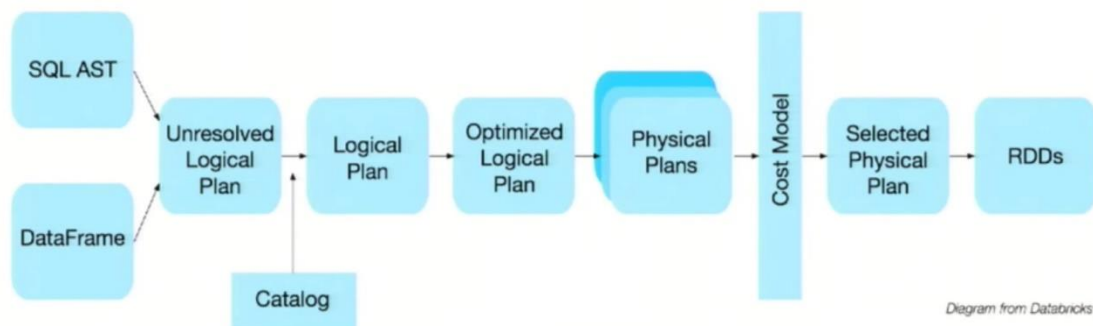
## Alte prestazioni

I DataFrame offrono prestazioni molto superiori ai RDD per due motivi:

1. Gestione dedicata della memoria (aka *Project Tungsten*)

I dati sono immagazzinati in forma binaria nella memoria off-heap (quindi non sottoposti a Garbage Collection. Viene anche evitata la serializzazione Java)

2. Piani di esecuzione ottimizzati (aka *Catalyst Optimizer*)



<https://databricks.com/blog/2015/04/28/project-tungsten-bringing-spark-closer-to-bare-metal.html>  
<https://databricks.com/blog/2014/03/26/spark-sql-manipulating-structured-data-using-spark-2.html>

# Dataset

API strutturata e typesafe

- Disponibile solo in Scala e Java
- Controllo sul tipo durante la compilazione
- Usa gli *Encoder* per convertire i tipi *domain-specific* nei tipi interni di *Spark*
- La API dei Dataset cerca di ottenere il meglio dei due mondi:
  - Orientata agli oggetti e type-safe
  - Mantiene le ottimizzazioni *Catalyst* e *Tungsten*

```
import spark.implicits._
case class Beer(id:Integer, name:String, ounces:Double)
val ds = beers.as[Beer]
ds.filter(beer=>beer.ounces>12.0).show()
```

_c0	abv	ibu	id	name	style	brewery_id	ounces
50	0.065	null	2603	Galaxyfest	American IPA	27	16.0
51	0.05	45.0	2602	Citrafest	American IPA	27	16.0
52	0.09	null	2220	Barn Yeti	Belgian Strong Da...	27	16.0
53	0.069	65.0	2219	Scarecrow	American IPA	27	16.0

# DataFrame vs Dataset

Tipizzazione dinamica vs statica

**DataFrame** (il tipo è noto durante l'esecuzione)

```
import org.apache.spark.sql.types._  
  
val schema = StructType(Array(  
  StructField("name", StringType),  
  StructField("age", IntegerType)))
```

Schema noto  
solo a run time

```
val df = spark.read.schema(schema).csv("../data/test.csv")  
  
spark.sql.DataFrame = [name: string, age: int]
```

*df.first()* è di  
tipo **Row**.

Il tipo della prima  
"Colonna" è noto  
solo a run time

```
df.first()(0)
```

```
res50: Any = Gian Marco  
Gian Marco
```

Non posso invocare il  
metodo *toUpperCase()* su  
una variabile di tipo **Any**  
(a compilation time)

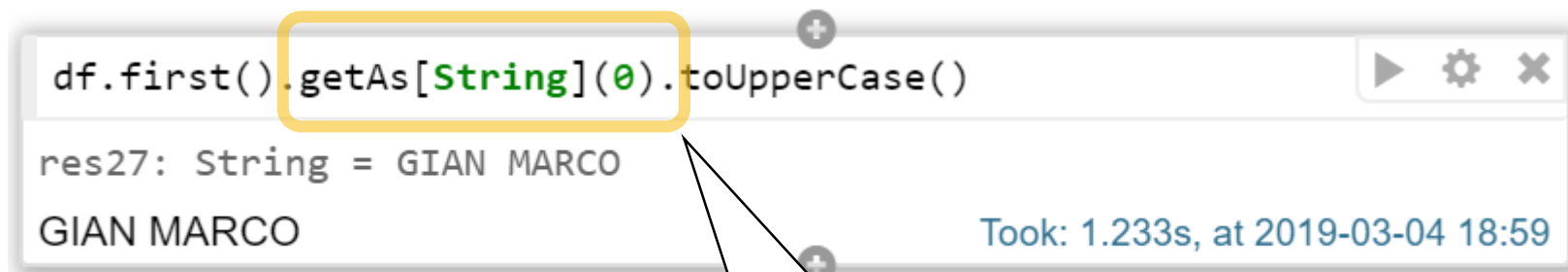
```
df.first()(0).toUpperCase()
```

```
<console>:89: error: value toUpperCase is not a member of Any  
      df.first()(0).toUpperCase()  
                        ^
```

# DataFrame vs Dataset

Tipizzazione dinamica vs statica

**DataFrame** (il tipo è noto durante l'esecuzione)



```
df.first().getAs[String](0).toUpperCase()
```

res27: String = GIAN MARCO  
GIAN MARCO

Took: 1.233s, at 2019-03-04 18:59

Devo fare un *cast*  
a *String*



# DataFrame vs Dataset

Tipizzazione dinamica vs statica

**Dataset[T]** (il tipo è noto durante la compilazione)

```
import spark.implicits._  
case class Person(name: String, age: Int)
```

Al posto dello schema  
usiamo una case class.  
Nota durante la  
compilazione

```
val ds = df.as[Person]
```

```
ds: org.apache.spark.sql.Dataset[Person] = [name: string, age: int]
```

Took: 2.632s, at 2019-03-04 19:34

```
ds.first().name.toUpperCase()
```

```
res68: String = GIAN MARCO  
GIAN MARCO
```

Posso invocare  
il metodo **!!!**  
`toUpperCase()`

`ds.first()` è di  
tipo **Person**.

Name è di tipo String  
(noto durante la  
compilazione)

# Conversioni

RDD  $\Rightarrow$  DataFrame

Inferendo lo schema tramite la reflection:

```
import spark.implicits._
val df = rdd.toDF()
// specificando i nomi delle colonne
val df2 = rdd.toDF("name", "age")
```

Specificando lo schema esplicitamente:

```
import org.apache.spark.sql.types._
val mySchema = StructType(Array(
    StructField("name", StringType),
    StructField("age", IntegerType)))
var rddRow = rdd.map(r=>Row(r._1, r._2))
val df3 = spark.createDataFrame(rddRow, mySchema)
```



*RDD[Row]*

# Conversioni

RDD  $\Rightarrow$  Dataset

```
val ds = spark.createDataset(rdd)
```

Oppure

```
case class Person(name:String, age:Int)
var ds = spark.createDataset(
    rdd.map(r=>Person(r._1,r._2)))
```

# Conversioni

DataFrame  $\Rightarrow$  Dataset

```
import spark.implicits._  
case class Person(name:String, age:Int)  
val ds = df.as[Person]
```

Dataset  $\Rightarrow$  DataFrame

```
val df = ds.toDF()
```

# Conversioni

DataFrame/Dataset  $\Rightarrow$  RDD

Non crea un nuovo RDD, ma accede a quello associato al DataFrame / Dataset

```
val rdd1 = df.rdd
```

*RDD[Row]*

```
val rdd2 = ds.rdd
```

*RDD[Person]*

*Dataset[Person]*

# DataFrame

Elementi chiave di un DataFrame:

- Colonne & espressioni
- Schema
- Righe (o record)
- Trasformazioni & Azioni

## DataFrame : Colonne

Sono molto simili alle colonne in un **foglio di calcolo** oppure in un **database**.

In Spark ci si può riferire ad una colonna in diversi modi:

```
col("name")  
column("name")  
df.col("name")  
df("name")  
$"name"  
" 'name"
```

L'effettiva esistenza della colonna viene determinata nella fase di analisi.

Il metodo *columns* permette di ottenere i nomi di tutte le colonne di un *DataFrame*:

```
df.columns  
  
res21: Array[String] = Array(id, date, time, lat, long, country, city, area, direction, dist, depth, x  
m, md, richter, mw, ms, mb)
```

## DataFrame : Espressioni

Gli RDD e I Dataset possono essere manipolati facilmente con l'operatore map().

```
// rdd[Int], Dataset[Int]
rdd1.map(x=>x*3) ds1.map(x=>x*3)

// rdd[(Double, Double)], Dataset[Point]
rdd2.map { case (x,y) =>(x,y*y) }
ds2.map(p=>Point(p.x, p.y*p.y))
```



## DataFrame : Espressioni

Le *espressioni* permettono di selezionare, manipolare e aggiungere colonne.

Un'*espressione* è un insieme di trasformazioni che agiscono su uno o più valori sui record di un DataFrame.

```
val df2 = df.select($"lat" + $"long")
```

```
df2: org.apache.spark.sql.DataFrame = [(lat + long): double]
```

Took: 0.853s, at 2019-02-28 11:23

```
val df3 = df.select(expr("lat + long"))
```

```
df3: org.apache.spark.sql.DataFrame = [(lat + long): double]
```

Took: 0.730s, at 2019-02-28 11:23

# DataFrame : Espressioni

```
earthquakes.withColumn("severity",
    when($"richter" > 6.0, "strong")
    .when($"richter" > 4.0, "medium")
    .otherwise("normal"))
.select("date","city","richter","severity")
.withColumn("date", $"date".cast("String"))
.take(10)
```

```
res14: Array[org.apache.spark.sql.Row] = Array([1966-08-19,mus,4.7,medium], [1966-08-19,mus,4.7,medium], [1966-12-30,sakarya,4.3,medium], [1967-05-22,mugla,4.6,medium], [1974-01-26,burdur,4.0,normal], [2017-02-06,canakkale,3.6,normal], [2014-10-21,izmir,4.1,medium], [1967-08-14,sakarya,4.2,medium], [1970-04-10,kutahya,4.2,medium], [1970-04-10,kutahya,4.2,medium])
```



10 entries total

Show:

10 ▼

Search:

<u>date</u>	<u>city</u>	<u>richter</u>	<u>severity</u>
1966-08-19	mus	4.7	medium
1966-08-19	mus	4.7	medium
1966-12-30	sakarya	4.3	medium
1967-05-22	mugla	4.6	medium
1974-01-26	burdur	4	normal
2017-02-06	canakkale	3.6	normal
2014-10-21	izmir	4.1	medium

## DataFrame : lo schema

Una lista di informazioni per ogni colonna: nome, tipo ed eventuali metadati.

```
df.printSchema

root
|-- id: decimal(3,-11) (nullable = true)
|-- date: string (nullable = true)
|-- time: string (nullable = true)
|-- lat: double (nullable = true)
|-- long: double (nullable = true)
|-- country: string (nullable = true)
|-- city: string (nullable = true)
|-- area: string (nullable = true)
|-- direction: string (nullable = true)
|-- dist: double (nullable = true)
|-- depth: double (nullable = true)
|-- xm: double (nullable = true)
|-- md: double (nullable = true)
|-- richter: double (nullable = true)
|-- mw: double (nullable = true)
|-- ms: double (nullable = true)
|-- mb: double (nullable = true)

Took: 0.924s, at 2019-02-28 11:34
```

Spark usa un sistema di tipi dedicati gestiti da un ottimizzatore chiamato *Catalyst*.

# DataFrame : lo schema

Lo schema puo' essere gestito proceduralmente:

df.schema

```
res59: org.apache.spark.sql.types.StructType = StructType(StructField(id,DecimalType(3,-11),true), StructField(date,StringType,true), StructField(time,StringType,true), StructField(lat,DoubleType,true), StructField(long,DoubleType,true), StructField(country,StringType,true), StructField(city,StringType,true), StructField(area,StringType,true), StructField(direction,StringType,true), StructField(dist,DoubleType,true), StructField(depth,DoubleType,true), StructField(xm,DoubleType,true), StructField(md,DoubleType,true), StructField(richter,DoubleType,true), StructField(mw,DoubleType,true), StructField(ms,DoubleType,true), StructField(mb,DoubleType,true))
```

```
import org.apache.spark.sql.types._
val mioSchema = StructType(Array(
  StructField("id",IntegerType,true),
  StructField("name", StringType, true)))
```

```
import org.apache.spark.sql.types._
mioSchema: org.apache.spark.sql.types.StructType = StructType(StructField(id,IntegerType,true), StructField(name,StringType,true))
```

Took: 0.813s, at 2019-02-28 11:45

## DataFrame : Righe

Le righe corrispondono ai vari record.

In un *DataFrame* le righe sono istanze della classe *Row*

```
df.first()
```

```
res51: org.apache.spark.sql.Row = [2.00E+13,2003.05.20,12:17:44 AM,39.04,40.38,turkey,bingol,baliklica  
y,west,0.1,10.0,4.1,4.1,0.0,null,0.0,0.0]
```

```
[2.00E+13,2003.05.20,12:17:44 AM,39.04,40.38,turkey,bingol,baliklicay,west,0.1,10.0,4.1,4.1,0.0,null,0.0,0.0]
```

Took: 0.833s, at 2019-02-28 11:27

```
df.first().getString(5)
```

```
res55: String = turkey
```

```
turkey
```

Took: 0.968s, at 2019-02-28 11:28

## DataFrame : Trasformazioni + Azioni



Spark Operations =

+



ACTIONS

# DataFrame : Trasformazioni

Le trasformazioni permettono di:

- Aggiungere/Rimuovere Righe/Colonne
- Trasformare Righe in Colonne e viceversa
- Cambiare l'ordine delle righe in base ai valori delle colonne

Sono eseguite in modo “pigro”.

Sono gli archi del *DAG* che rappresenta l'elaborazione da eseguire.

# DataFrame : Trasformazioni

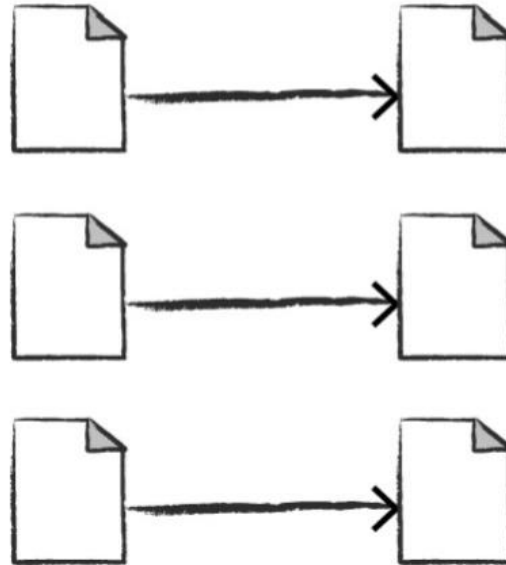
Metodi:

*select(), selectExpr(), lit(), withColumn(), withColumnRenamed(),  
drop(), filter(), where(), distinct(), sample(), randomSplit(),  
union(), orderBy()...*



## DataFrame : Trasformazioni

Narrow transformations  
1 to 1

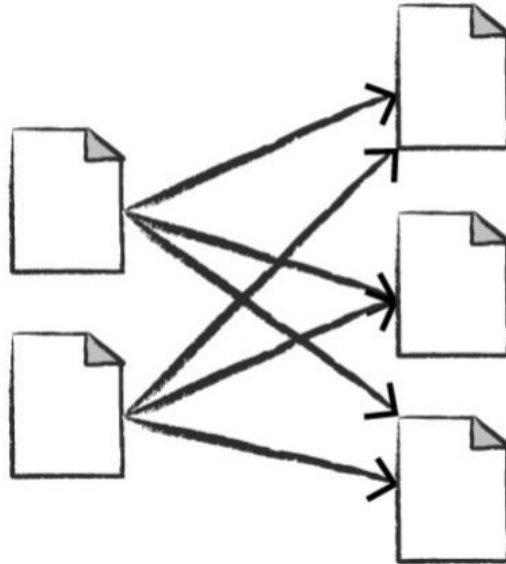


Es:

*select(), selectExpr(), lit(), withColumn(), withColumnRenamed(), drop(), filter(), where(), sample(), randomSplit()*

## DataFrame : Trasformazioni

Wide transformations  
(shuffles) 1 to N



Es:

*`distinct()`, `sort()`, `join()`, `orderBy()`*

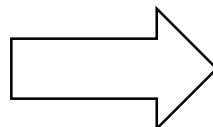
# DataFrame : Trasformazioni

## Narrow

```
val df2 = df1.select($"x", $"y" + 1 as "y")
```

Partizione 1

x	y	z
64	89	47
28	30	55
22	42	22
...	...	...

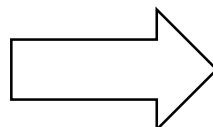


Partizione 1

x	y
64	90
28	31
22	43
...	...

Partizione 2

27	12	51
50	79	28
30	47	85
...	...	...



Partizione 2

27	13
50	80
30	48
...	...

# DataFrame : Trasformazioni

Wide

```
val df3 = df2.sort("x")
```

Partizione 1

x	y
64	90
28	31
22	43
...	...

Partizione 2

27	13
50	80
30	48
...	...

Partizione 1

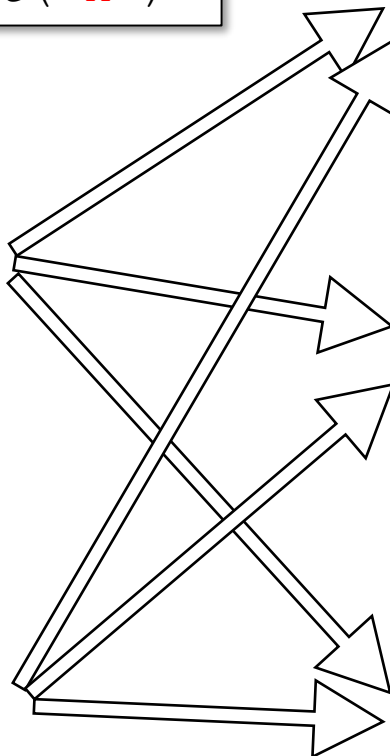
x	y
50	80
64	90
...	...

Partizione 2

x	y
22	43
27	13
...	...

Partizione 3

28	31
30	48
...	...

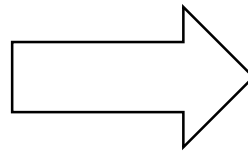


# DataFrame : Trasformazioni

## explode

```
lines.select($"index",explode(split($"line", " ")))
```

index	line
0	nel mezzo del cammin di nostra vita
1	mi ritrovai per una selva oscura
2	che la diritta via era smarrita
3	ahi quanto a dir qual era e cosa dura
4	esta selva selvaggia e aspra e forte
5	che nel pensier rinova la paura
6	tant e amara che poco e piu morte
7	ma per trattar del ben ch i vi trovai
8	diro de l altre cose ch i v ho scorte
9	io non so ben ridir com i v intrain
...	...



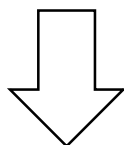
Index	word
0	nel
0	mezzo
0	del
0	cammin
0	di
0	nostra
0	vita
1	mi
1	ritrovai
1	per
...	...

# DataFrame : Trasformazioni

## explode

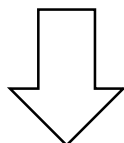
```
lines.select($"index",explode(split($"line", " ")))
```

nel mezzo del cammin di nostra vita



split()

Array(nel, mezzo, del, cammin, di, nostra, vita)



explode()

nel
mezzo
del
cammin
di
nostra
vita

# DataFrame : Azioni

Le azioni innescano il calcolo

Ci sono tre tipi di azioni:

- Visualizzare dati nella console
- Raccogliere dati in oggetti nativi nel linguaggio
- Scrivere l'output nello storage

Esempi: *count()*, *show()*, *collect()*, *first()*, *take()*, *reduce()*, *countApproxDistinct()*, *max()*, *min()*

## DataFrame : Azioni

Ogni azione viene eseguita da un *job* Spark che:

- Esegue le trasformazioni *narrow* (es. filtri)
- Esegue le aggregazioni (trasformazioni *wide*)
- Raccoglie i dati in un oggetto nativo nel linguaggio utilizzato (es. Scala).



# Data Source

# Data Sources

Come arrivano i dati dentro Spark?

Spark gestisce sei tipi fondamentali di *data source*:

- CSV (*Comma Separated Values*)
- JSON (*JavaScript Object Notation*)
- Parquet
- ORC (*Apache Optimized Row Columnar*)
- Connessioni JDBC/ODBC
- Testo semplice

# Data Sources

Come arrivano i dati dentro Spark?

La comunità crea innumerevoli altri *data source*:

- Cassandra
- HBase
- MongoDB
- AWS Redshift
- XML
- Ecc.



## Il formato CSV (Comma Separated values)

Dati tabulari in un semplice file di testo

Year	Make	Model	Description	Price
1997	Ford	E350	ac, abs, moon	3000.00
1999	Chevy	Venture "Extended Edition"		4900.00
1999	Chevy	Venture "Extended Edition, Very Large"		5000.00
1996	Jeep	Grand Cherokee	MUST SELL! air, moon roof, loaded	4799.00

```
Year,Make,Model,Description,Price
1997,Ford,E350,"ac, abs, moon",3000.00
1999,Chevy,"Venture ""Extended Edition""",,4900.00
1999,Chevy,"Venture ""Extended Edition, Very Large""",,5000.00
1996,Jeep,Grand Cherokee,"MUST SELL! air, moon roof, loaded",4799.00
```

# DataFrameReader

L'API per leggere un Dataset (DataFrame)

Per accedere al *DataFrameReader* si usa *spark.read*. Es:

```
val df = spark.read.csv("prova.csv")
```

La struttura base dei comandi di lettura è:

```
DataFrameReader.format(...).option("key", "value").schema(...).load()
```

Questa struttura permette di specificare diversi valori:

- Il formato (es. «csv»)
- Lo schema
- La modalità di lettura (come reagire agli errori)
- Una serie di opzioni (che possono dipendere dal formato)

# DataFrameReader

L'API per leggere un Dataset (DataFrame)

Un esempio:

```
val df = spark.read.format("csv")  
  .option("mode", "FAILFAST")  
  .option("inferSchema", "true")  
  .option("path", "/path/to/file")  
  .schema(ilmioSchema)  
  .load()
```

# DataFrameReader

L'API per leggere un Dataset (DataFrame)

Modo di lettura (opzione “mode”):

- *Permissive*
- *dropMalformed*
- *failFast*

# Schema manuale vs. inferito

Cos'è meglio?

Dipende dal contesto.

Per le analisi ad-hoc lo schema inferito va in genere bene

In produzione si preferisce definire lo schema a priori.



# DataFrameWriter

L'API per scrivere un Dataset (DataFrame)

Per accedere al *DataFrameWriter* si usa *dataFrame.write*. Es:

```
df.write.option("header", true).csv("prova.csv")
```

Attenzione! Vengono creati tanti file quante sono le partizioni. Se si vuole un file solo:

```
df.coalesce(1).write.option("header", true).csv("prova.csv")
```

# Date, Stringhe e altri grattacapi

## Date e Timestamps

Se Spark non riesce a riconoscere il formato:

```
withColumn("date", to_date($"date", "yyyy.MM.dd"))
```

Confronti e altre operazioni:

```
filter($"date" < "2000-01-01")
```

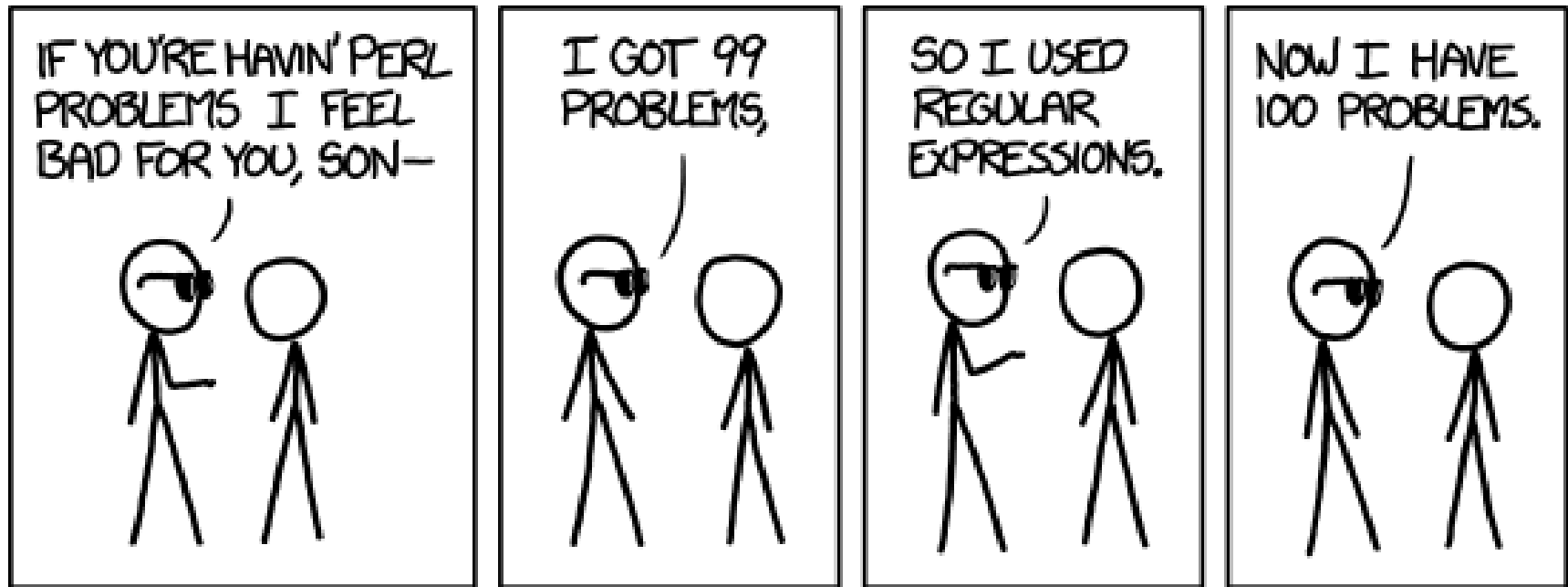
```
withColumn("diff", datediff($"date", lit("2000-01-01")))
```

# Stringhe

Sono disponibili molte funzioni di manipolazione delle stringhe. Ad es.:

- `initcap()`, `lower()`, `upper()`
- `ltrim()`, `rtrim()`, `trim()`, `lpad()`, `rpadd()`
- `contains()`, `startsWith()`

## Regular Expression in Spark



[https://imgs.xkcd.com/comics/perl\\_problems.png](https://imgs.xkcd.com/comics/perl_problems.png)

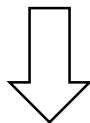
# Regular expression in Spark

Sono disponibili due funzioni:

- `regexp_extract()`
- `regexp_replace()`

```
val cleanedText = text.withColumn("line",  
    regexp_replace($"line", "[^a-zA-Z]", " "))
```

"Parole, parole, ...!"



"Parole parole "

## Gestione dei *null*

`coalesce(column)`

Restituisce i valori non nulli

`df.na.drop()`

Elimina le righe con valori nulli

`df.na.fill(val)`

Mette val al posto dei valori nulli

# UDF

Definire le proprie trasformazioni

```
def pow3(x:Integer):Integer = x*x*x
val pow3udf = udf(pow3(_:Integer):Integer)

df.select($"x", pow3udf($"x").as("x^3"))
```



## Combinare due DataFrame

Il metodo *union* concatena due dataframe

```
val df3 = df1.union(df2)
```

1. Non fa una vera union: nel risultato ci possono essere record duplicati (che possono essere eliminati facendo un successivo *distinct*).
2. Le colonne vengono unite in base all'ordine, non al nome.
3. È a cura del programmatore assicurarsi che gli schemi dei due dataframe coincidano.

## Combinare due DataFrame

È possibile fare una *join* fra due dataframe.

```
val df3 = df1.join(df2, expression, joinType)
```

1. I tipi di join supportati sono: *inner*, *cross*, *outer*, *full*, *full\_outer*, *left*, *left\_outer*, *right*, *right\_outer*, *left\_semi*, *left\_anti*
2. Il default è *inner*.
3. Attenzione a *cross*!! (Prodotto cartesiano)

Qualche esempio:

```
val df3 = df1.join(df2, df1.col("name") === df2.col("ref")) // inner
val df4 = df1.join(df2, df1.col("name") === df2.col("ref"), "full")
val df5 = df1.crossJoin(df2)
```

# SQL

Puoi trasformare un DataFrame in una tabella SQL:

```
df.createOrReplaceTempView("table")
```

E poi dare dei comandi SQL:

```
val modoSql = spark.sql("""
select city,count(1) from earthquake
where richter>0
group by city
""")

val modoScala = earthquake
  .select("city", "richter")
  .where("richter > 0").groupBy("city").count()

modoSql.explain
modoScala.explain
```