

BIG DATA & ANALYTICS

SQL OVER BIG DATA - SPARK SQL

Pierluca Ferraro

pierluca.ferraro@unipa.it

Università degli Studi di Palermo



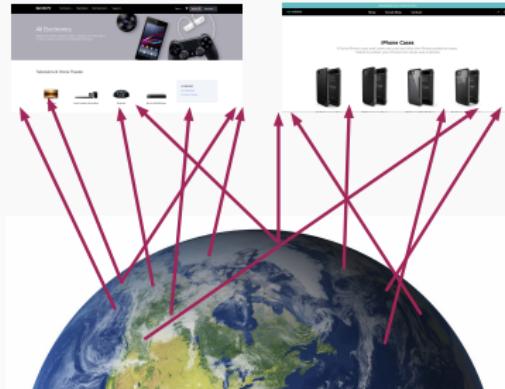
BIG DATA ANALYTICS

PERCHÉ API DI ALTO LIVELLO?

- Gli RDD sono adatti per rappresentare **coppie chiave-valore**, ma diventano scomodi se abbiamo a che fare con tanti campi.
- Operazioni comuni come il **join** di più dataset sono meno immediate di quanto dovrebbero essere.
- Ad esempio, si devono spesso creare RDD temporanei con una nuova chiave, al solo scopo di effettuare un join.
- Calcolare più operazioni di aggregazione (ad esempio minimo, massimo, media) richiede spesso di effettuare operazioni ripetitive.
- All'interno di Spark, le API degli RDD sono considerate di basso livello.
- API di alto livello, che permettono di eseguire **query SQL** sui dati, consentono anche ai non programmati di analizzare i dati.

ESEMPIO - ANALISI DEI LOG DI UN SERVER WEB

- Abbiamo a disposizione 4 dataset eterogenei:
 - log di un server Web;
 - database di geolocalizzazione;
 - dati personali degli utenti;
 - lista di bot noti e utenti fake.
- Ciascuno di questi dataset potrebbe essere disponibile in un formato diverso: CSV, JSON, Common Log Format, tabella di un database...
- Che tipo di analisi possiamo effettuare?



Web-service access logs



user personal data



Geobase



[ro]bot database

ACCESS LOG IN BASE AL PAESE DI PROVENIENZA



Web-service access logs



Geobase

ACCESS LOG IN BASE AL PAESE DI PROVENIENZA



Web-service access logs

Geobase

- Unendo i log di un server Web con dati provenienti da un database di geolocalizzazione, è possibile verificare il Paese di provenienza degli utenti.
- Questi dati possono essere utilizzati, ad esempio, per decidere se è conveniente tradurre un sito Web in una particolare lingua, per effettuare analisi di mercato o per offrire servizi personalizzati a una determinata fetta di utenza.

ACCESS LOG IN BASE AL PAESE DI PROVENIENZA



Web-service access logs

Geobase

- Esempio di una riga di log:

```
109.169.248.247 - - [12/Dec/2015:18:25:11 +0100] "GET /administrator/ HTTP/1.1" 200 4263 "-"  
"Mozilla/5.0 (Windows NT 6.0; rv:34.0) Gecko/20100101 Firefox/34.0" "-"
```

- Informazioni presenti: indirizzo IP, timestamp, pagina Web richiesta, response code, user agent...
- Esempio di una riga di un database di geolocalizzazione:
`"109.169.128.0", "109.170.127.255", "RU"`
- Significato della riga: tutti gli indirizzi IP compresi tra 109.169.128.0 e 109.170.127.255 provengono dalla Russia.
- Effettuando un join tra questi due dataset possiamo facilmente calcolare i Paesi da cui provengono la maggior parte delle richieste.

CALCOLARE LA PERCENTUALE DI UTENTI REALI



Web-service access logs

[ro]bot database

- Unendo i log del server Web con una lista di bot noti, è possibile separare gli utenti reali da quelli fintizi.
- In questo caso si utilizzano sia gli indirizzi IP sia gli user agent.
- Ovviamente, una semplice lista statica non è sufficiente per individuare bot e account fake con sicurezza.
- Spesso vengono utilizzate tecniche di machine learning per aumentare la precisione dei sistemi di identificazione degli account fake.



Web-service
access logs



Geobase



User personal
data

- Avendo a disposizione anche un database di utenti, è possibile effettuare altri tipi di analisi, che tengano in considerazione l'età media dei clienti, la distribuzione di utenti uomini e donne...
- Tutte queste informazioni possono essere messe in relazione alla provenienza geografica, al tipo di prodotto e così via.
- È importante che informazioni di questo genere possano essere calcolate in modo semplice ed efficiente, anche lavorando con grandi moli di dati.

SPARK SQL - INTRODUZIONE

SPARK SQL DATA FLOW

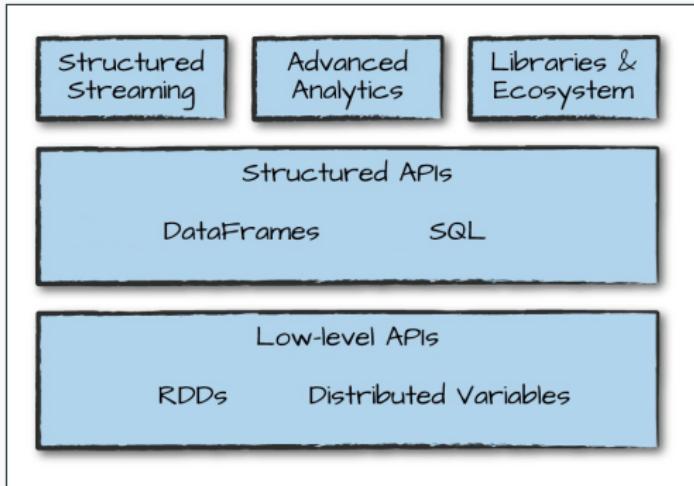


- Spark SQL è l'interfaccia di Spark per lavorare con dati strutturati e semistrutturati.
- Può leggere dati da una varietà di fonti strutturate (comprese tabelle Hive e database esterni).

- Spark SQL fornisce una ricca integrazione tra SQL e il normale codice Python/Java/Scala, inclusa la possibilità di effettuare join tra RDD e tavole SQL, definire funzioni personalizzate in SQL e altro ancora.
- Per implementare queste funzionalità, Spark SQL fornisce un tipo speciale di RDD chiamato **DataFrame**.



ARCHITETTURA DI SPARK



- DataFrame e SQL sono le API distribuite di Spark.
- Internamente, tutto si basa sulle operazioni tra RDD.
- A partire da Spark 2.0, le API di Spark SQL sono quelle consigliate da Apache per la maggior parte degli utilizzzi.

SORGENTI DATI SUPPORTATE

- Come Hive, anche Spark SQL consente di lavorare con sorgenti di dati di ogni genere:

built-in



{ JSON }



external



|

and more ...



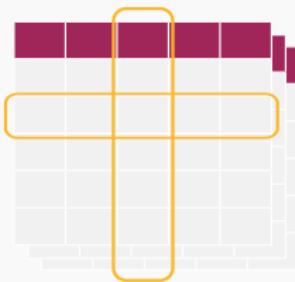
JDBC - JAVA DATABASE CONNECTIVITY

- In particolare, utilizzando JDBC, viene garantita l'interoperabilità con i principali database relazionali.



SPARK SQL - DATAFRAME

CHE COS'È UN DATAFRAME?

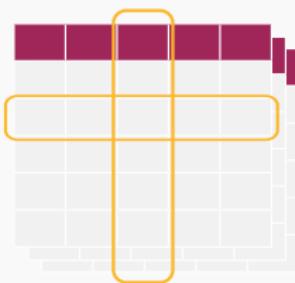


DataFrame = RDD + schema

- Un DataFrame è un RDD di oggetti appartenenti alla classe **Row**, ciascuno dei quali rappresenta un **record**.
- Un DataFrame è caratterizzato anche da uno **schema** (nomi e tipi dei campi) delle sue righe.
- Esempio:

```
>>> sharing_df.printSchema()
root
|-- date: timestamp (nullable = true)
|-- casual: integer (nullable = true)
|-- registered: integer (nullable = true)
|-- total_users: integer (nullable = true)
```

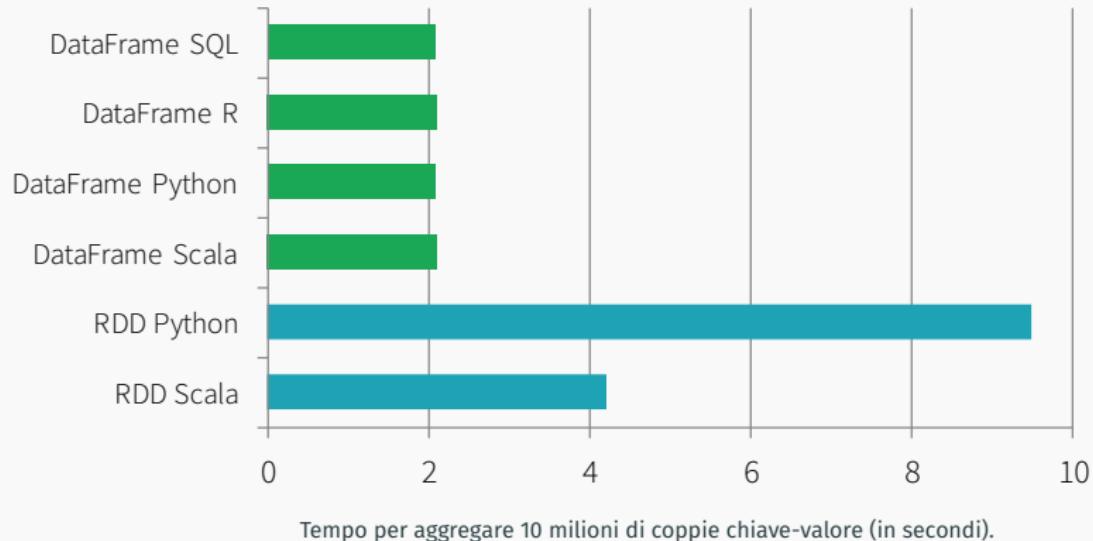
CHE COS'È UN DATAFRAME?



DataFrame = RDD + schema

- I DataFrame assomigliano a RDD normali, ma internamente memorizzano i dati in modo **più efficiente**, sfruttando il loro schema.
- Inoltre, consentono nuove operazioni non disponibili sui normali RDD, come la possibilità di **eseguire query SQL**.
- I DataFrame possono essere creati a partire da fonti di dati esterne, da risultati di query precedenti o da normali RDD.

PERFORMANCE DI SPARK SQL



- I DataFrame possono essere molto più veloci degli RDD, per alcuni tipi di query, e le loro prestazioni non dipendono dal linguaggio utilizzato.

- I DataFrame sono implementati sulla base degli RDD, per cui è possibile utilizzare le normali trasformazioni e azioni degli RDD.
- La proprietà `rdd` di un DataFrame consente di accedere in ogni momento all'RDD corrispondente.
- In aggiunta, Spark fornisce metodi aggiuntivi per creare DataFrame a partire da moltissime sorgenti dati, come file JSON, XML o database.
- Lo schema di un DataFrame può essere specificato manualmente, oppure **inferito** automaticamente da Spark.
- Prima di utilizzare il metodo `sql()`, è necessario **registrare** un DataFrame come tabella, utilizzando il metodo `createOrReplaceTempView()`.
- Tabelle di questo genere sono **temporanee**, e vengono cancellate al termine della sessione pyspark.

CREAZIONE DI DATAFRAME A PARTIRE DA RDD

- Un RDD costituito da righe di tipo Row può essere convertito in DataFrame utilizzando il metodo `toDF()`.
- In alternativa, si possono creare oggetti di tipo Row a partire da dizionari:

```
from pyspark.sql import Row

# Supponiamo che sharing sia un RDD composto da dizionari
sharing.take(1)    # [{"casual": 331, "date": "2011-01-01",
                      #   "registered": 654, "tot_users": 985}]

sharing_df = sharing.map(lambda dictionary: Row(**dictionary)).toDF()
```

- Spark inferisce automaticamente lo schema:

```
sharing_df.printSchema()
# root
# |-- casual: long (nullable = true)
# |-- date: string (nullable = true)
# |-- registered: long (nullable = true)
# |-- tot_users: long (nullable = true)
```

CREAZIONE DI DATAFRAME A PARTIRE DA FILE JSON

- Un DataFrame può essere creato in modo molto semplice a partire da file JSON, specificando il percorso (in locale o su HDFS) di un file o di una directory:

```
tweets_df = spark.read.json('tweets/', multiLine=True)
tweets_df.count()    # 10000
```

- Anche in questo caso Spark inferisce automaticamente lo schema:

```
tweets_df.printSchema()
# root
#   |-- created_at: string (nullable = true)
#   |-- favorite_count: long (nullable = true)
#   |-- hashtags: array (nullable = true)
#     |-- element: string (containsNull = true)
#   |-- id: string (nullable = true)
#   ...
```

- Spark 1.6 si aspetta di trovare un record JSON in ciascuna riga. Spark 2 accetta anche file JSON in cui gli elementi occupano più righe, se si specifica l'opzione `multiline=True`.

CREAZIONE DI UN DATAFRAME A PARTIRE DA UN DATABASE RELAZIONALE

- Per utilizzare un RDBMS come sorgente di dati è necessario scaricare il driver JDBC appropriato e indicare il suo path quando si fa partire pyspark, tramite l'opzione --jars:

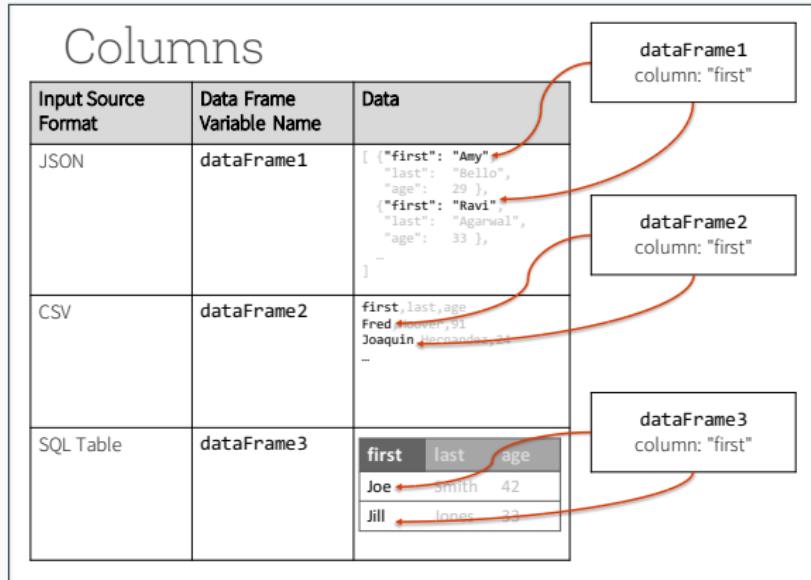
```
pyspark --jars "$HOME/BigData/Libraries/sqlite-jdbc-3.21.0.jar"
```

- Sarà poi possibile creare nuovi DataFrame associati a tabelle del database. Esempio, utilizzando SQLite:

```
df = (spark.read
      .options(driver='org.sqlite.JDBC')
      .jdbc(url='jdbc:sqlite:dbip.sqlite', table='ipcountry'))

df.printSchema()
# root
# |-- start: string (nullable = true)
# |-- end: string (nullable = true)
# |-- country: string (nullable = true)
```

COLONNE IN UN DATAFRAME



- Le colonne in un DataFrame sono un'astrazione che permette di accedere ai campi di interesse in modo **uniforme**, indipendentemente dalla sorgente di dati.

Principali tipi di dato in SPARK SQL, con i loro equivalenti in Python

| Spark SQL Type | Python Type | Data Type |
|-------------------------|-------------|---------------|
| INT | int, long | IntegerType |
| LONG | long | LongType |
| FLOAT | float | FloatType |
| STRING | string | StringType |
| BOOLEAN | bool | BooleanType |
| TIMESTAMP | datetime | TimestampType |
| ARRAY<TYPE> | list | ArrayType |
| MAP<KEY_TYPE, VAL_TYPE> | dict | MapType |
| ... | ... | ... |

REGISTRAZIONE DI UN DATAFRAME COME TABELLA

- Per poter eseguire query SQL, il DataFrame deve essere registrato come tabella, utilizzando il metodo `createOrReplaceTempView()`:

```
tweets_df = spark.read.json('tweets/')
tweets_df.createOrReplaceTempView('tweets')
```

- Quando un DataFrame è stato registrato, è possibile utilizzare il metodo `sql()` della SparkSession per eseguire una query:

```
query = spark.sql("""
SELECT *
FROM tweets
WHERE retweet_count > 10
""")
```

- Il metodo `sql()` restituisce un nuovo DataFrame, sul quale possono essere eseguite ulteriori operazioni.

QUERY SQL

- Come nel caso degli RDD, le query vengono valutate in modo **lazy**, solo quando si esegue un'azione.
- L'azione `show(n)` mostra i primi `n` risultati della query:

```
query = spark.sql("""  
SELECT created_at, text, retweet_count  
FROM tweets  
WHERE retweet_count > 10  
ORDER BY retweet_count DESC  
""")  
  
query.show(5)  
# +-----+-----+  
# | created_at | text | retweet_count |  
# +-----+-----+  
# | 2018-05-05 16:59:27 | We're getting rea... | 564 |  
# | 2018-05-05 13:32:56 | Which Machine Lea... | 474 |  
# | 2018-05-03 21:02:40 | #Robots & #Hu... | 248 |  
# | 2018-05-01 19:44:21 | The Moments Innov... | 222 |  
# | 2018-05-04 14:23:04 | Human-centered De... | 206 |  
# +-----+-----+  
# only showing top 5 rows
```

QUERY SQL

- Sono supportati i principali costrutti di SQL, tra cui:
 - `SELECT`
 - `FROM`
 - `WHERE`
 - `JOIN` (inner, left, right...)
 - `GROUP BY`
 - `HAVING`
 - `ORDER BY`
 - `AS` (alias)
 - `LIMIT`
 - ...
- Spark 1.6 supporta le query annidate solo all'interno delle clausole `FROM`.
- Spark 2 supporta anche query annidate all'interno delle clausole `WHERE`.

UDF - USER DEFINED FUNCTIONS

UDF - USER DEFINED FUNCTIONS

- Spark SQL consente di utilizzare funzioni scritte in Python/Scala/Java all'interno delle query SQL.
- Per registrare una **User Defined Function** si usa il metodo
`spark.udf.register()`
- È necessario specificare anche il tipo di ritorno della funzione. I possibili tipi sono quelli mostrati nella tabella della slide 27 (pag. 35), colonna "Data Type".
- Il tipo di default, se non specificato, è `StringType()`.
- Esempio:

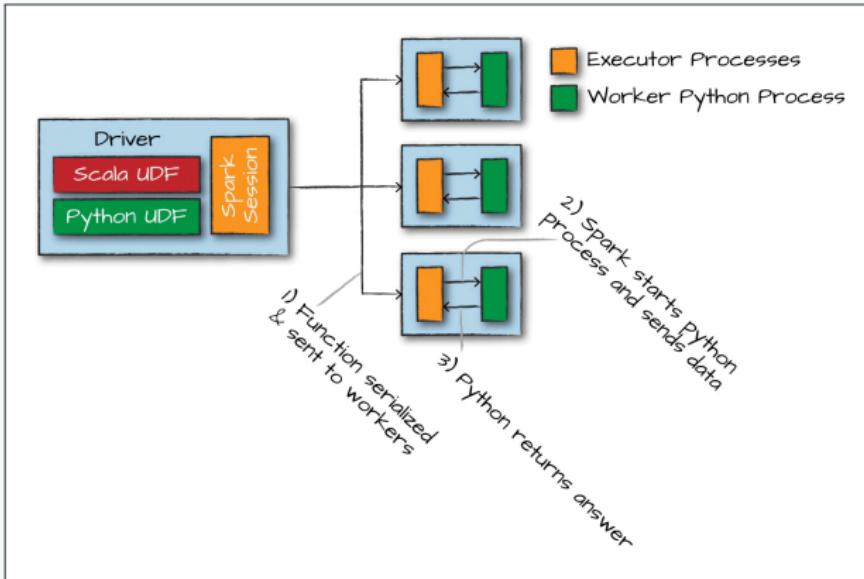
```
from pyspark.sql.types import FloatType

def square(number):
    return float(number * number)

spark.udf.register('square', square, FloatType())

spark.sql('''
SELECT retweet_count, square(retweet_count) AS squared
FROM tweets
''').show()
```

PYTHON UDF



- Funzionamento delle UDF scritte in Python.
- Fare partire il processo Python nei nodi executor e, soprattutto, passare i dati da e verso Python, è costoso.

UDF - COLONNE MULTIPLE

- Normalmente le funzioni UDF lavorano su una sola colonna.
- Se è necessario passare i valori di più colonne alla UDF, si può usare il tipo complesso **ARRAY**:

```
def sum_cols(columns):
    return sum(columns)

spark.udf.register('sum_cols', sum_cols, IntegerType())

spark.sql('''
SELECT *, sum_cols(ARRAY(casual, registered)) AS casual_registered
FROM sharing
''' ).show()
```

- In Java e Scala è possibile definire anche **UDAF** (User Defined Aggregation Functions).

OPERAZIONI SUGLI ARRAY

OPERAZIONI SUGLI ARRAY

- Per lavorare con il tipo complesso ARRAY sono previste diverse funzioni.
- Tra queste, le più interessanti sono:
 - `split`
 - `explode`
 - `array_contains`
 - `collect_list`
 - `size`

ESEMPIO SPLIT

- `split` divide una stringa in base ad un carattere separatore e restituisce un ARRAY:
- Esempio - dividere il testo dei Tweet in parole:

```
SELECT split(`text`, " ") as words  
FROM tweets
```

```
+-----+  
|       words |  
+-----+  
|[#Cybersecurity, ...|  
|[That, part, wher...|  
|[Implementing, a,...|  
|[The, Efficiency,...|  
|[The, latest, The...|  
|[#Tesla, #shareho...|  
|[4, reasons, for,...|  
+-----+  
only showing top 7 rows
```

ESEMPIO EXPLODE

- explode accetta come argomento il nome di una colonna che contiene un ARRAY e crea una nuova riga (con gli altri campi duplicati) per ogni valore contenuto nell'ARRAY.
- Esempio - dividere la lista di hashtag presenti in un tweet (la colonna hashtags della tabella tweets è di tipo ARRAY):

```
SELECT explode(hashtags) as hashtag  
FROM tweets
```

```
+-----+  
|      hashtag|  
+-----+  
| Cybersecurity|  
| AutonomousVehicles|  
| Insurtech|  
| BigData|  
| AI|  
| opendata|  
| smartcities|  
+-----+  
only showing top 7 rows
```

ESEMPIO ARRAY_CONTAINS

- `array_contains` restituisce un valore booleano che indica se un valore è presente o meno in un ARRAY.
- Esempio - verificare se un particolare hashtag è presente:

```
SELECT hashtags, array_contains(hashtags, "BigData") as contains_bigdata
FROM tweets
```

```
+-----+-----+
|       hashtags | contains_bigdata
+-----+-----+
|[Cybersecurity, A....|      true
|[opendata, smartc...|      false
|[Python, abdsc, B....|      true
|[BigData]|      true
|[ai, bigdata]|      false
|[Tesla, sharehold...|      false
|[automotive, IIoT...|      false
+-----+-----+
only showing top 7 rows
```

ESEMPIO COLLECT_LIST

- `collect_list` raggruppa diversi valori in un ARRAY.
- Esempio - temperatura minima, massima e lista del meteo per ogni stagione

```
SELECT season, min(temp) AS min_temp, max(temp) AS max_temp,
       collect_list(weather) AS weather
  FROM meteo
 GROUP BY season
```

| season | min_temp | max_temp | weather |
|--------|----------|----------|-----------------------|
| 1 | 2.421 | 23.471 | [2, 2, 1, 1, 1, 1...] |
| 2 | 10.371 | 33.141 | [2, 1, 2, 2, 1, 1...] |
| 3 | 19.241 | 35.331 | [2, 1, 2, 1, 1, 1...] |
| 4 | 9.051 | 26.961 | [2, 2, 2, 2, 2, 2...] |

ESEMPIO SIZE

- size restituisce il numero di elementi presenti in un ARRAY.
- Esempio - contare il numero di hashtag presenti in un tweet

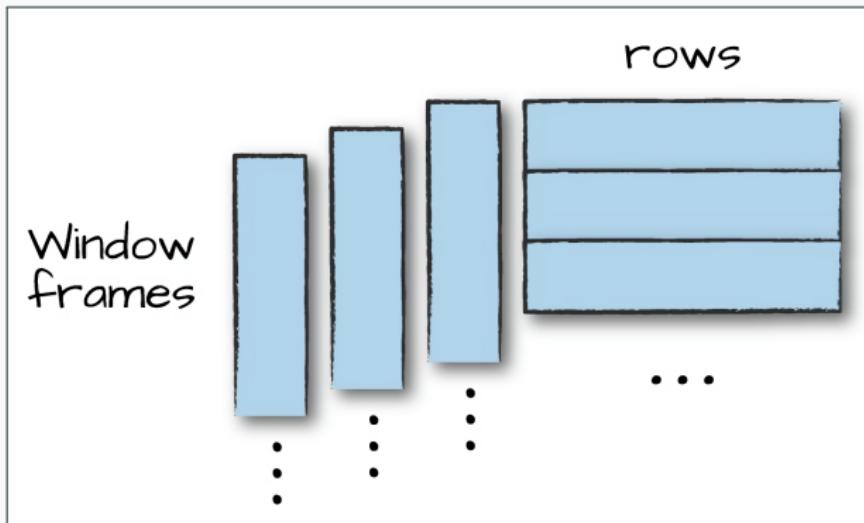
```
SELECT hashtags, size(hashtags) AS num_hashtags  
FROM tweets
```

```
+-----+-----+  
|       hashtags | num_hashtags |  
+-----+-----+  
|[Cybersecurity, A...]|      5|  
|[opendata, smartc...]|      4|  
|[Python, abdsc, B...]|      8|  
|      [BigData] |      1|  
|      [ai, bigdata] |      2|  
|[Tesla, sharehold...]|     23|  
|[automotive, IIoT...]|      5|  
+-----+-----+  
only showing top 7 rows
```

WINDOW FUNCTIONS

- Utilizzando solo funzioni di aggregazione e UDF, alcuni tipi di operazioni sono difficili (se non impossibili) da esprimere.
- In particolare, non è possibile operare su un gruppo di righe (come con le funzioni di aggregazione) e contemporaneamente restituire un valore per ogni riga di input (come con le UDF).
- Questa limitazione rende difficile il calcolo di alcune quantità utilizzate spesso per l'analisi dei dati, come la **media mobile**, la **somma cumulativa** o l'accesso ai valori di una **riga che compare prima** (o dopo) la riga corrente.
- Un esempio di media mobile potrebbe essere il calcolo della temperatura media considerando solo le ultime N letture sensoriali.
- La somma cumulativa è utile per calcolare, ad esempio, il saldo di un conto corrente, dopo ogni operazione.

WINDOW FUNCTIONS



- Una window function calcola un valore di ritorno per ogni riga di input di una tabella, in base ad un gruppo di righe, chiamato **frame**.
- Più frame possono contenere, al loro interno, la stessa riga.

TIPI DI WINDOW FUNCTION

- Spark SQL supporta tre tipi di window function:
 - funzioni di aggregazione;
 - funzioni di ranking;
 - funzioni analitiche.
- Le funzioni di aggregazione sono le stesse comunemente usate insieme a `GROUP BY`: `COUNT`, `SUM`, `MIN`, `MAX`, `Avg` e così via.
- Le funzioni di ranking più comuni sono: `RANK`, `DENSE_RANK` e `ROW_NUMBER`.
- Le funzioni analitiche più comuni sono: `FIRST_VALUE`, `LAST_VALUE`, `LAG` e `LEAD`.
- Per utilizzare una window function si deve definire la finestra su cui deve operare, introdotta dalla parola chiave `OVER`. Esempio:

```
SELECT *, AVG(temp) OVER (<definizione finestra>)
FROM ...
WHERE ...
```

DEFINIZIONE DI UNA FINESTRA

- La definizione di una finestra prevede di specificare una o più colonne in base alle quali raggruppare i dati (`PARTITION BY`), un ordinamento (`ORDER BY`) e un frame (`ROWS BETWEEN`).
- `PARTITION BY` controlla quali righe saranno nello stesso gruppo (in modo simile a `GROUP BY`). Se non viene indicato, si intende l'intera tabella.
- `ORDER BY` controlla il modo in cui sono ordinate le righe **all'interno di ogni gruppo**.
- `ROWS BETWEEN` indica quali righe saranno incluse nel frame per la riga di input corrente, in base alla loro posizione relativa rispetto alla riga corrente stessa.
 - Ad esempio, `ROWS BETWEEN 3 PRECEDING AND CURRENT ROW` descrive un frame che include la riga corrente e le tre righe precedenti.

ESEMPIO CON UNA FUNZIONE DI AGGREGAZIONE

- Calcolare la temperatura media stagionale, per ogni riga:

```
SELECT *, AVG(temp) OVER (PARTITION BY season) AS avg_temp  
FROM meteo
```

```
+-----+-----+-----+-----+-----+  
|      date | hum | season | templ | weather | avg_temp |  
+-----+-----+-----+-----+-----+  
| 2011-01-01 | 80.58 | 1 | 14.11 | 2 | 12.21 |  
| 2011-01-02 | 69.61 | 1 | 14.91 | 2 | 12.21 |  
| 2011-01-03 | 43.73 | 1 | 8.05 | 1 | 12.21 |  
| 2011-01-04 | 59.04 | 1 | 8.21 | 1 | 12.21 |  
+-----+-----+-----+-----+-----+  
only showing top 4 rows
```

- Tutte le righe vengono mantenute.
- Viene aggiunta una nuova colonna, `avg_temp`, con la temperatura media della stagione corrispondente a ciascuna riga.

ESEMPIO CON UNA FUNZIONE DI AGGREGAZIONE

- Calcolare la temperatura media stagionale, per ogni riga:

```
SELECT *, AVG(temp) OVER (PARTITION BY season) AS avg_temp
FROM meteo
+-----+-----+-----+-----+
|      date|  hum|season| templ|weather| avg_temp|
+-----+-----+-----+-----+
|2011-01-01|80.58|    1|14.11|     2|       12.21|
|2011-01-02|69.61|    1|14.91|     2|       12.21|
|2011-01-03|43.73|    1| 8.05|     1|       12.21|
|2011-01-04|59.04|    1| 8.21|     1|       12.21|
+-----+-----+-----+-----+
only showing top 4 rows
```

- Utilizzando `GROUP BY` avremmo ottenuto una sola riga per gruppo:

```
SELECT season, AVG(temp) AS avg_temp
FROM meteo
GROUP BY season
+-----+
|season| avg_temp|
+-----+
|    1|   12.21|
|    2|   22.32|
|    3|   28.96|
|    4|   17.34|
+-----+
```

ESEMPIO CON MIN, MAX, AVG

- Calcolare la differenza tra la temperatura di ciascun giorno e la temperatura stagionale massima, minima e media:

```
SELECT date, temp,
       temp - AVG(temp) OVER w AS diff_avg,
       temp - MAX(temp) OVER w AS diff_max,
       temp - MIN(temp) OVER w AS diff_min
  FROM meteo
 WINDOW w AS (PARTITION BY season)
```

| | date | temp | diff_avg | diff_max | diff_min |
|---|------------|-------|----------|----------|----------|
| 1 | 2011-01-01 | 14.11 | 1.90 | -9.36 | 11.69 |
| 2 | 2011-01-02 | 14.90 | 2.69 | -8.57 | 12.48 |
| 3 | 2011-01-03 | 8.05 | -4.16 | -15.42 | 5.63 |
| 4 | 2011-01-04 | 8.21 | -4.01 | -15.27 | 5.78 |
| 5 | 2011-01-05 | 9.31 | -2.90 | -14.16 | 6.89 |

only showing top 5 rows

- Sintassi alternativa per definire una finestra (`WINDOW w AS (<definizione>)`), in modo da poterla utilizzare più volte all'interno della query.

FUNZIONI DI RANKING

- `ROW_NUMBER` restituisce il numero di riga all'interno di ogni gruppo.
- `RANK` è simile a `ROW_NUMBER` ma assegna lo stesso rank alle righe con lo stesso valore (come in una classifica).

```
SELECT RANK() OVER (ORDER BY num_requests DESC) AS ranking, code, country,
       num_requests
  FROM logs_country
```

| ranking | code | country | num_requests |
|---------|------|--------------------|--------------|
| 1 | RU | Russian Federation | 298981 |
| 2 | US | United States | 195671 |
| 3 | AT | Australia | 146181 |
| 3 | UA | Ukraine | 146181 |
| 5 | KZ | Kazakhstan | 53751 |
| 6 | DE | Germany | 37581 |

only showing top 6 rows

- `DENSE_RANK` è come `RANK`, ma rende la classifica **compatta** (nell'esempio avrebbe restituito 4 per il Kazakhstan e 5 per la Germania).

FUNZIONI ANALITICHE

- `LAG(column)` e `LEAD(column)` restituiscono il valore della colonna specificata nella riga precedente (o successiva) rispetto a quella corrente.
- `FIRST_VALUE(column)` e `LAST_VALUE(column)` restituiscono il valore della colonna specificata nella prima (o ultima) riga del gruppo.
- Esempio - Link della richiesta attuale e precedente di un utente:

```
SELECT ip_addr, request, LAG(request) OVER w AS prev_request  
FROM web_logs  
WINDOW w AS (PARTITION BY ip_addr ORDER BY unix_time)
```

| ip_addr | request | prev_request |
|-----------------|------------------------|------------------------|
| 131.108.164.124 | /apache-log/access.log | null |
| 131.108.164.124 | /wp-login.php | /apache-log/access.log |
| 131.108.164.124 | ?action=register | /wp-login.php |
| 188.138.56.91 | /robots.txt | null |
| 188.138.56.91 | / | /robots.txt |

only showing top 5 rows

COME SPECIFICARE IL FRAME

- Per specificare quali righe includere nel frame della riga corrente, si usa la clausola `ROWS BETWEEN <expr1> AND <expr2>`.
- I possibili valori delle espressioni sono:
 - `UNBOUNDED PRECEDING`
 - `UNBOUNDED FOLLOWING`
 - `CURRENT ROW`
 - `<num_rows> PRECEDING`
 - `<num_rows> FOLLOWING`
- `UNBOUNDED PRECEDING` e `UNBOUNDED FOLLOWING` rappresentano, rispettivamente, la prima e l'ultima riga del gruppo corrente.
- `CURRENT ROW` rappresenta la riga corrente.
- `<num_rows> PRECEDING` e `<num_rows> FOLLOWING` rappresentano, rispettivamente, `<num_rows>` prima e dopo la riga corrente.

ESEMPI DI FRAME

Visual representation of frame
ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING

Current input row =>

| product | category | revenue |
|------------|------------|---------|
| Bendable | Cell phone | 3000 |
| Foldable | Cell phone | 3000 |
| Ultra thin | Cell phone | 5000 |
| Thin | Cell phone | 6000 |
| Very thin | Cell phone | 6000 |

≤ 1 PRECEDING

≤ 1 FOLLOWING

- ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW seleziona tutte le righe dall'inizio del gruppo fino a quella corrente.
- ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING seleziona tutte le righe del gruppo.
- ROWS BETWEEN 3 PRECEDING AND CURRENT ROW seleziona, all'interno del gruppo, le tre righe precedenti più la riga corrente.

ESEMPIO - SOMMA CUMULATIVA

- Calcolare il saldo di un conto corrente, dopo ogni transazione (ovvero la somma di tutte le operazioni compiute da un utente, dall'inizio fino alla riga corrente):

```
SELECT id, user_id, operazione, SUM(operazione) OVER w AS saldo
FROM transactions
WINDOW w AS (PARTITION BY user_id ORDER BY id
               ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
```

| id | user_id | operazione | saldo |
|-----|---------|------------|-------|
| I1 | I2 | -2.27 | -2.27 |
| I10 | I2 | 10.55 | 8.28 |
| I13 | I2 | 4.17 | 12.45 |
| I14 | I2 | 11.21 | 23.66 |
| I20 | I2 | -14.36 | 9.30 |
| I21 | I2 | 8.19 | 17.49 |
| I22 | I2 | -4.23 | 13.26 |

only showing top 7 rows

ESEMPIO - MEDIA MOBILE

- Calcolare la media degli ultimi 3 valori di temperatura (compreso quello corrente):

```
SELECT date, temp, AVG(temp) OVER w AS moving_avg
FROM meteo
WINDOW w AS (ORDER BY date ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)
```

| date | temp | moving_avg |
|------------|-------|------------|
| 2011-01-01 | 14.11 | 14.11 |
| 2011-01-02 | 14.91 | 14.50 |
| 2011-01-03 | 8.05 | 12.35 |
| 2011-01-04 | 8.21 | 10.38 |
| 2011-01-05 | 9.31 | 8.52 |
| 2011-01-06 | 8.38 | 8.63 |
| 2011-01-07 | 8.06 | 8.58 |
| 2011-01-08 | 6.77 | 7.74 |
| 2011-01-09 | 5.67 | 6.83 |
| 2011-01-10 | 6.18 | 6.21 |

only showing top 10 rows