Big Data

Spark

Argomenti della lezione

- Cos’è Spark
- L’ecosistema Spark
- PySpark
- Resilient Distributed Datasets (RDDs)
- Esercitazione

Cos’è Spark

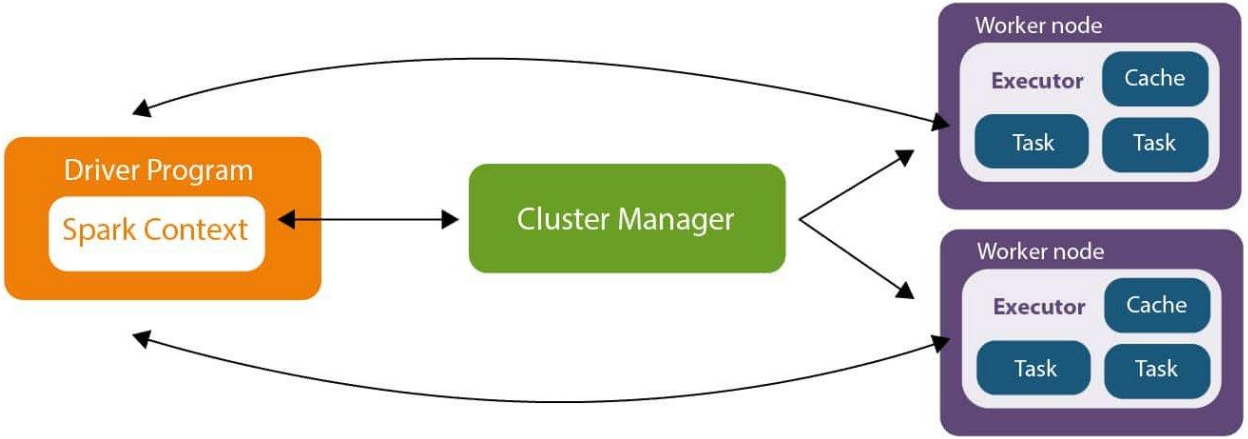
Apache Spark è un framework di elaborazione dei dati basato su una generalizzazione di MapReduce.

Spark è stato progettato per essere tanto un framework indipendente quanto un’API per lo sviluppo di applicazioni. Il sistema è studiato per effettuare un’elaborazione dei dati general purpose in memoria e per eseguire dei flussi di lavoro di stream e una computazione interattiva e iterativa.

L’architettura di Spark è centrata attorno al concetto di RDD (Resilient Distribuited Dataset), una raccolta di oggetti di sola lettura suddivisi su un set di macchine che possono persistere in memoria. Formalmente si tratta di una collezione di record partizionata in sola lettura.

Un’applicazione Spark è costituita da un programma driver che esegue operazioni parallele su un cluster di processi worker a lunga durata e che possono conservare le partizioni dei dati in memoria passando funzioni che vengono eseguite come attività parallele.

I processi sono coordinati attraverso un’istanza di SparkContext. Questo si connette a un gestore delle risorse (come YARN), richiede gli esecutori sui nodi worker e invia le attività che devono essere eseguite. Gli esecutori sono i responsabili dell’esecuzione delle attività e della gestione della memoria a livello locale.



Spark consente di condividere le variabili tra le attività, o tra queste ed il driver, usando l'astrazione delle variabili condivise. Di queste ne supporta due tipi: le variabili di broadcast, che possono essere usate per conservare un valore in memoria su tutti i nodi, e gli accumulatori, variabili aggiuntive come contatori e somme.

Resilient Distribuited Dataset (RDD)

Un RDD viene salvato in memoria, condiviso tra le macchine ed è usato nelle operazioni parallele come quelle di MapReduce. La tolleranza dei guasti viene ottenuta attraverso il *lineage* (la "discendenza"): se una partizione di una RDD va perduta, l'RDD prende dagli altri RDD le informazioni sull'origine sufficienti a poter ricostruire esattamente quella partizione.

Un RDD può essere costruito in quattro modi:

- leggendo i dati da un file salvato in HDFS;

- suddividendo una collezione in un dato numero di partizioni che vengono inviate ai worker;
- trasformando un RDD esistente utilizzando operatori paralleli;
- modificando la persistenza di un RDD esistente.

Spark dà il meglio di se quando gli RDD si adattano alla memoria e possono essere conservati tra le operazioni. L'API espone i metodi per la persistenza degli RDD e consente numerose strategie di persistenza e livelli di storage, permettendo lo spill su disco oltre che la serializzazione binaria per l'efficienza dello spazio.

Azioni

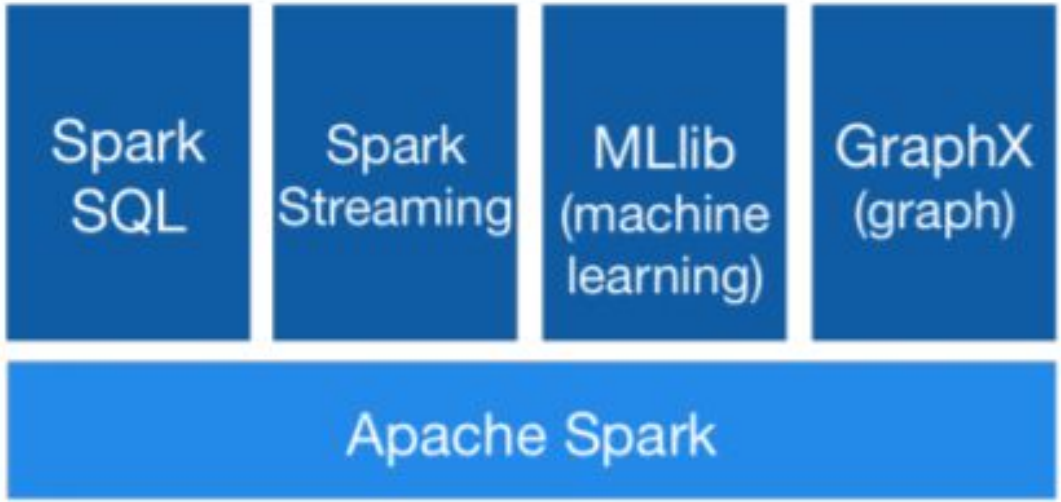
Le operazioni vengono invocate passando le funzioni a Spark, il sistema gestisce le variabili e gli effetti collaterali secondo il paradigma di programmazione funzionale, Le closure possono fare riferimento alle variabili nell'ambito in cui sono state create. Esempi di azioni sono **count** (che restituisce il numero di elementi nel dataset) e **save** (che genera il dataset nello storage).

Altre operazioni parallele sugli RDD:

- **map**: applica una funzione a ciascun elemento del dataset;
- **filter**: seleziona gli elementi da un dataset secondo criteri forniti dall'utente;
- **reduce**: combina gli elementi di un dataset usando una funzione associativa;
- **collect**: invia tutti gli elementi del dataset al programma driver;
- **foreach**: passa ogni elemento attraverso una funzione fornita dall'utente;
- **groupByKey**: raggruppa le voci in base a una chiave;
- **sortByKey**: ordina le voci per chiave.

L’ecosistema Spark

Apache Spark fornisce diversi strumenti, sia come libreria sia come motore di esecuzione.



Spark Streaming
Spark Streaming è un'estensione dell'API Scala che contiene l'elaborazione dei dati da streaming come i socket Kafka, Flume, Twitter, ZeroMQ e TCP.

Spark Streaming riceve stream di dati di input live e li divide in batch (finestre temporali dimensionate in modo arbitrario) che vengono poi elaborati dal motore core di Spark per generare lo stream finale dei risultati. Questa astrazione di alto livello è chiamata DStream ed è implementata come una sequenza di RDD. DStream rende possibili due tipi di operazioni: quelle di trasformazione e quelle di output.

Le trasformazioni lavorano su uno o più DStream per crearne di nuovi. Come parte della catena delle trasformazioni, i dati possono essere resi persistenti a un livello di storage (HDFS) oppure su un canale di output. Spark Streaming consente trasformazioni in una finestra scorrevole di dati.
Un'operazione basata su finestre richiede la specifica di tre parametri: la lunghezza della finestra, la sua durata e l'intervento di scorrimento in base al quale l'operazione verrà eseguita.

GraphX

GraphXè una libreria per l'analisi di grafi talmente grandi che non potrebbero essere analizzati su una singola macchina (ad esempio i grafi dei social network). La libreria offre algoritmi come PageRank (per misurare quanto è "importante" ogni nodo di un grafo), il calcolo delle componenti connesse, il calcolo dei triangoli, ecc..

MLlib

MLlib fornisce funzionalità Machine Learning (ML) comuni, tra cui test e generazione di dati. MLlib supporta attualmente quattro tipi di algoritmi: classificazione binaria, regressione, clustering e filtro collaborativo.

SparkSQL

Spark SQL deriva da Shark, un'implementazione del sistema di warehousing di Hive che utilizza Spark come motore di esecuzione. Con Spark SQL è possibile mescolare query di tipo SQL con codice Scala o Python. I set di risultati restituiti da una query sono degli RDD, e in quanto tali possono essere manipolati dai metodi principali di Spark o da MLlib e GraphX.

PySpark

PySpark è l'API Python per Spark, essa ci consente di creare applicazioni Spark da una shell interattiva o tramite applicazioni Python.

Prima di eseguire qualsiasi istruzione Python all'interno di Spark, è necessario creare un oggetto **SparkContext** che indica a Spark come e dove accedere a un cluster.

La proprietà **master** è l'URL del cluster che determina dove verrà eseguita l'applicazione. I principali valori utilizzati per la proprietà **master** sono i seguenti:

- **local** esegue Spark con un unico worker thread;
- **local[n]** esegue Spark con n worker thread;
- **spark://HOST:PORT** connessione ad un cluster Spark standalone
- **mesos://HOST:PORT** connessione ad un cluster mesos

Shell Interattiva

Nella shell Spark l'oggetto **SparkContext** è creato quando la shell viene lanciata. Il riferimento allo SparkContext creato viene assegnato alla variabile **sc**.

All'avvio della console il parametro **master,** se non passato, verrà valorizzato di default con il valore **local[*]**. Per tanto il nostro SparkContext punterà all'istanza locale di Spark e potrà utilizzare un numero variabile di worker thread.

Il parametro **master** per la shell interattiva può essere settato utilizzando l'argomento **--master** con il comando di avvio della shell **pyspark**.

$ pyshell --master local[4]

```
$ pyspark --master local[4]
...
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 1.5.0
      /_/
Using Python version 2.7.10 (default, Jul 13 2015 12:05:58)
SparkContext available as sc, HiveContext available as sqlCon-
text.
>>>
```

Per una lista completa delle opzioni è possibile eseguire il comando **pyspark --help**

Naturalmente, oltre ad eseguire del codice all'interno della shell interattiva, è possibile creare delle applicazioni Spark in Python che possono essere eseguite autonomamente.

Queste applicazioni, non avendo il supporto della shell, devono creare un oggetto **SparkContext** prima di poter utilizzare un qualsiasi metodo di Spark.

Il master può essere impostato alla creazione dell'oggetto SparkContext come segue:

```
sc = SparkContext(master='local[4]')
```

Per eseguire un'applicazione Spark scritta in Python si deve utilizzare lo script **spark-submit**.

```
$ spark-submit --master local spark_app.py
```

Anche questo comando prevede diverse opzioni che sono consultabili eseguendo il comando **spark-submit --help**.

WordCount in PySpark

Il codice seguente implementa l'algoritmo di WordCount in PySpark. Esso assume che all'interno del path **/user/cloudera/datasets/** sia stato caricato il file input.txt mentre l'output generato sarà salvato nel path, sempre HDFS, **/user/cloudera/spark/wc-out**

```
from pyspark import SparkContext
def main():
    sc = SparkContext(appName='SparkWordCount')
    input_file = sc.textFile('/user/cloudera/datasets/input.txt')
    counts = input_file.flatMap(lambda line: line.split()) \
        .map(lambda word: (word, 1)) \
        .reduceByKey(lambda a, b: a + b)
    counts.saveAsTextFile('/user/cloudera/spark/wc-out')
    sc.stop()

if __name__ == '__main__':
    main()
```

Di seguito è riportata una breve descrizione dell'algoritmo implementato nella funzione main.

La prima istruzione crea un oggetto di tipo SparkContext. Questo oggetto dice a spark dove e come contattare il cluster da utilizzare per le elaborazioni:

```
sc = SparkContext(appName='SparkWordCount')
```

La seconda istruzione utilizza lo SparkContext per caricare il file da HDFS e ne salva un riferimento nella variabile input_file

```
input_file = sc.textFile('/user/cloudera/datasets/input.txt')
```

La terza istruzione applica diverse trasformazioni sui dati di input. Spark parallelizza automaticamente queste trasformazioni per l'esecuzione su più macchine

```
counts = input_file.flatMap(lambda line: line.split()) \
                   .map(lambda word: (word, 1)) \
                   .reduceByKey(lambda a, b: a + b)
```

La quarta istruzione si occupa di salvare il risultato delle trasformazioni in un path HDFS:

```
counts.saveAsTextFile('/user/cloudera/spark/wc-out')
```

La quinta ed ultima istruzione si occupa di terminare lo SparkContext.

```
sc.stop()
```

Resilient Distributed Datasets (RDDs)

I Resilient Distributed Datasets sono delle collezioni immutabili di dati, partizionate tra le macchine, che consentono di eseguire le operazioni sugli elementi in parallelo.

Gli RDD possono essere costruiti in diversi modi: parallelizzando una collection Python esistente, referenziando un files in un sistema di archiviazione esterno come HDFS oppure effettuando delle trasformazioni agli RDD esistenti.

Creazione da collection

Come anticipato gli RDD possono essere creati da una collection Python chiamando il metodo SparkContext.parallelize(). Gli elementi della collection sono copiati per formare un set di dati distribuito che può essere utilizzato in parallelo.

L'esempio seguente crea una raccolta parallelizzata da una lista:

```
>>> data = [1, 2, 3, 4, 5]
>>> rdd = sc.parallelize(data)
>>> rdd.glom().collect()
[[1, 2, 3, 4, 5]]
```

Il metodo RDD.glom() restituisce la lista di tutti gli elementi presenti in tutte le partizioni. Mentre il metodo RDD.collect() restituisce la lista di tutti gli elementi presenti in questo RDD. L'ultima riga mostra come sono state splittate (parallelizzate) le informazioni.

E' possibile specificare il numero di partizioni in cui frazionare i valori presenti all'interno del nostro RDD. Per farlo è sufficiente passare, come secondo parametro della funzione parallelize, un intero che indicherà il numero di partizioni:

```
>>> rdd = sc.parallelize(data, 4)
>>> rdd.glom().collect()
[[1], [2], [3], [4, 5]]
```

Creare un RDD da una risorsa esterna

Gli RDD possono anche essere creati da file esterni utilizzando il metodo **SparkContext.textFile()**.

Spark può leggere i file dal filesystem locale, ma anche da qualsiasi altro sistema di archiviazione supportato da Hadoop, Amazon S3 e così via.
Spark supporta file di testo, SequenceFiles, qualsiasi altro formato supportato da Hadoop, directory, file compressi e wildcards,, ad esempio, my/directory/*.txt.

L'esempio seguente crea un set di dati distribuito da un file situato nel filesystem locale:

```
>>> distFile = sc.textFile('data.txt')
>>> distFile.glom().collect()
[[u'jack be nimble', u'jack be quick', u'jack jumped over the candlestick']]
```

in questo caso il file è stato caricato in un’unica partizione. Allo stesso modo di parallelize, textFile accetta un secondo parametro che determina il numero di partizioni

RDD Operations

Gli oggetti di tipo RDD supportano 2 tipologie di operazioni: trasformazioni e azioni. Le **trasformazioni** creano un nuovo dataset a partire da quello esistente mentre le **azioni** eseguono una elaborazione su un dataset e restituiscono il risultato al programma driver.

Le trasformazioni sono di tipo LAZY: cioè il loro risultato non viene calcolato immediatamente. Spark ricorda tutte le trasformazioni applicate al dataset di base. Le trasformazioni sono calcolate quando un’azione richiede che venga restituito un risultato al programma driver.

Per impostazione predefinita, le trasformazioni possono essere ricalcolate ogni volta che un'azione viene rieseguita. Ciò consente a Spark di utilizzare in modo efficiente la memoria, ma può utilizzare più risorse di elaborazione se le stesse trasformazione vengono elaborate costantemente. Per garantire che una trasformazione sia calcolato una sola volta, l'RDD risultante può essere conservato in memoria utilizzando il metodo RDD.cache()

Workflow di un RDD

Il workflow generale per lavorare con RDD è il seguente:

1. Creare un RDD da un'origine dati;
2. Applicare le trasformazioni a un RDD;
3. Applicare azioni a un RDD.

Il seguente esempio utilizza questo workflow per calcolare il numero di caratteri in un file:

```
>>> lines = sc.textFile('data.txt')
>>> line_lengths = lines.map(lambda x: len(x))
>>> document_length = line_lengths.reduce(lambda x,y: x+y)
>>> print document_length
59
```

La prima istruzione crea un RDD dal file esterno data.txt. Il file non viene caricato qui ma viene attribuito alla variabile *lines* un riferimento alla fonte esterna.

La seconda istruzione esegue una trasformazione sulla RDD di base utilizzando la funzione **map()** per calcolare il numero di caratteri in ogni riga. La variabile line_lengths non viene calcolato immediatamente a causa della natura lazy delle trasformazioni.

Infine, viene chiamato il metodo **reduce()**, che è un'azione. A questo punto, Spark divide la computazione in task eseguiti su macchine separate. Ogni macchina esegue sia map() che reduce() sui suoi dati locali, restituendo i risultati al programma.

Se l'applicazione dovesse usare di nuovo la variabile line_lengths , sarebbe meglio renderla persistente per evitare che venga ricalcolata. La riga seguente salverà la variabile in memoria:

>>> line_lengths.persist()

Le funzioni Lambda in Python

Molte delle trasformazioni e delle azioni di Spark richiedono funzioni da passare dal programma driver per l'esecuzione nel cluster.

Il modo più semplice per definire e passare una funzione è attraverso l'uso di Funzioni lambda di Python.

Le funzioni Lambda sono funzioni anonime che vengono create in fase di esecuzione. L'esempio seguente mostra una funzione lambda che restituisce la somma dei suoi due argomenti:

```
lambda a, b: a + b
```

La parola chiave **lambda** crea una funzione inline che contiene una singola espressione. Il valore di questa espressione è ciò che la funzione restituisce quando viene richiamata.

Le Trasformazioni su RDD

Le trasformazioni creano nuovi set di dati da quelli esistenti. L'esecuzione lazy consente a Spark di ricordare l'insieme di trasformazioni applicate alla RDD di base consentendogli di ottimizzare i calcoli richiesti.

Di seguito sono riportate alcune delle trasformazioni più comuni di Spark.

Per un elenco completo delle trasformazioni, fare riferimento alla documentazione delle API Python:

http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD

MAP

La funzione **map(func)** restituisce un nuovo RDD applicando la funzione func passata in input ad ogni elemento della sorgente.

Il seguente esempio moltiplica ogni elemento dell'RDD sorgente per due:

```
>>> data = [1, 2, 3, 4, 5, 6]
>>> rdd = sc.parallelize(data)
>>> map_result = rdd.map(lambda x: x * 2)
>>> map_result.collect()
>>> [2, 4, 6, 8, 10, 12]
```

FILTER

La funzione **filter(func)** restituisce un nuovo RDD contenente i soli elementi per i quali, applicando la funzione in input func, viene restituito il valore True.

Il seguente esempio ritorna solo i numeri pari:

```
>>> data = [1, 2, 3, 4, 5, 6]
>>> rdd = sc.parallelize(data)
>>> filter_result = rdd.filter(lambda x : x%2 == 0)
>>> filter_result.collect()
[2, 4, 6]
```

DISTINCT

La funzione **distinct()** restituisce un nuovo RDD privo di duplicati.

Esempio:

```
>>> data = [1, 2, 3, 2, 4, 1]
>>> rdd = sc.parallelize(data)
>>> distinct_result = rdd.distinct()
>>> distinct_result.collect()
[4, 1, 2, 3]
```

FLATMAP

La funzione **flatMap()** è molto simile a map(), ma restituisce una versione più appiattita del risultato.

Il seguente esempio restituisce l'elemento originale e la sua potenza; vediamo prima il codice con **map()**:

```
>>> data = [1, 2, 3, 4]
>>> rdd = sc.parallelize(data)
>>> map_result = rdd.map(lambda x: [x, pow(x,2)])
>>> map_result.collect()
[[1,1],[2,4],[3,9],[4,16]]
```

ed ora lo stesso esempio con **flatMap()**

```
>>> data = [1, 2, 3, 4]
>>> rdd = sc.parallelize(data)
>>> flatmap_result = rdd.flatMap(lambda x: [x, pow(x,2)])
>>> flatmap_result.collect()
[1, 1, 2, 4, 3, 9, 4, 16]
```

Le Azioni su RDD

Le azioni fanno sì che Spark computi le trasformazioni. Dopo che le trasformazioni vengono calcolate sul cluster il risultato viene restituito al programma driver.

Di seguito alcune delle azioni più comuni di Spark. Per un elenco completo delle azioni, fare riferimento alla documentazione delle API Python**:**

http://spark.apache.org/docs/latest/api/python/pyspark.html#pyspark.RDD

REDUCE

La funzione **reduce()** aggrega elementi in un RDD usando una funzione che prende due argomenti e ne ritorna uno. La funzione utilizzata deve essere commutativa e associativa, assicurandosi che possa essere eseguita correttamente in parallelo.

Il seguente esempio restituisce il prodotto di tutti gli elementi:

```
>>> data = [1, 2, 3]
>>> rdd = sc.parallelize(data)
>>> rdd.reduce(lambda a, b : a*b)
6
```

TAKE

La funzione **take(n)** ritorna un array contenente i primi n elementi dell'RDD. Il seguente esempio restituisce in RDD contenente i primi due elementi:

```
>>> data = [1, 2, 3]
>>> rdd = sc.parallelize(data)
>>> rdd.take(2)
[1, 2]
```

COLLECT

La funzione **collect()** ritorna tutti gli elementi dell'RDD in un array.

```
>>> data = [1, 2, 3, 4, 5]
>>> rdd = sc.parallelize(data)
>>> rdd.collect()
[1, 2, 3, 4, 5]
```

NB: l'invocazione di **collect()** su insiemi molto grandi può portare il driver in out of memory; in questi casi si consiglia di utilizzarlo insieme al metodo **take()**:

```
>>> rdd.take(100).collect()
```

TAKEORDERED

La funzione **takeOrdered(n, func)** ritorna un array contenente i primi n elementi dell'RDD, nel loro ordine naturale oppure secondo la funzione func.

Il seguente esempio restituisce in RDD contenente i primi quattro elementi in ordine discendente:

```
>>> data = [6, 1, 5, 2, 4, 3]
>>> rdd = sc.parallelize(data)
>>> rdd.takeOrdered(4, lambda s: -s)
[6, 5, 4, 3]
```

Esercitazione

Scrivere un programma Spark in Python che, a partire dalla lista di film presenti nel file **movies**, restituisca i titoli dei films che hanno al loro interno una determinata parola (il controllo non deve essere case sensitive).

Testare il risultato restituito anche tramite query in Hive.

Sul dasetset fornito, la ricerca della parola **batman** dovrebbe restituire il seguente risultato:

Batman (1989)
Batman & Robin (1997)
Batman Forever (1995)
Batman Returns (1992)