SABD 2022: Batch processing. Analisi del dataset dei taxi di NYC

ELENA SOFIA RUZZETTI, Universitá degli studi di Roma Tor Vergata, Laurea Magistrale in Informatica

1 ARCHITETTURA DEL CLUSTER

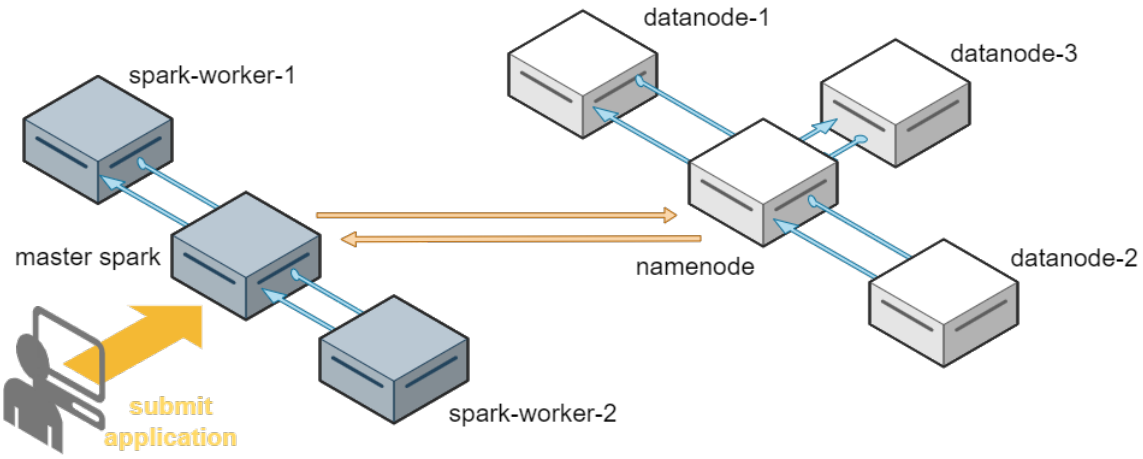

Fig. 1. Architettura dei servizi dell'applicazione

Si è predisposto un piccolo ecosistema per l'elaborazione in-memory di big-data. Per cercare di simulare un ambiente distribuito sulla macchina locale, i diversi servizi dell'applicazione sono stati realizzati come container docker, messi in comunicazione mediante `docker compose`.

`docker compose` è un framework di orchestrazione di container che esegue i container su un singolo computer host: permette di definire ed eseguire applicazioni Docker multi-container, esponendo ciascun container come servizio e organizzadoli sotto una rete. Ciò consente di eseguire un certo numero di container sull'host, ma di organizzarli e isolarli nella rete che si è definito. Con `docker compose`, si utilizza un file yaml per configurare i servizi dell'applicazione e con il comando *docker up*, si creano e si avviano tutti i servizi della configurazione.

La configurazione qui descritta è specificata nel file `docker-compose.yml` nella cartella principale della repository.

1.1 Configurazione HDFS

Tre container sono stati allocati al file system distribuito HDFS. Uno di questi ha il ruolo di *namenode* e altri due svolgono invece il ruolo di *datanode*. La comunicazione tra i diversi container in questo caso è gestita da `docker compose`.

Nel file `docker-compose.yml` si è specificata la porta esposta dal *namenode* e si è indicato che, affinchè possa essere identificato sia dai *datanodes* che da Spark. Il namenode è raggiungibile dagli altri container sulla porta 9000. L'interfaccia web è raggiungibile dalla porta 9870. Entrambe la porte sono mappate in `docker-compose.yml` alle stesse porte dell'host locale.

1.2 Configurazione Spark

Anche in questo caso si sono predisposti 3 container: un *master* e tre *worker*. Ancora una volta, grazie all'ultilizzo di `docker-compose.yml` sono state settate solo le porte dei diversi container.

Il container master è raggiungibile sulla porta 7077, mentre la sua l'interfaccia web è interrogabile sulla porta 8080 e sulla 4040 per aver il dettaglio dei job in esecuzione.

2 CARICAMENTO DEI DATI SU HDFS

Si è scelto di caricare su HDFS i file parquet che costituiscono il dataset. In fase di caricamento è stato impostato anche il grado di replicazione. Ogni file, data la particolare efficienza di memorizzazione data da questo formato di file, corrisponde ad un solo blocco sul file system distribuito. La dimensione del blocco per il file system distribuito è stata lasciata a quella di default pari a 128 MB.

Il caricamento è reso possibile dagli appositi comandi. Sul container del namenode eseguiamo la `-put` dei file che compongono il dataset dopo aver creato la cartella di destinazione `"/input"`. Si setta anche il grado di replicazione pari a 3: ognuno dei tre datanode conterrà quindi 3 blocchi, uno per file di input.

3 PREPROCESSAMENTO E DATA INJECTION

Per la fase di preprocessamento e data injection si è utilizzata l'API di Sperk. In particolare, si è fatto uso della API `org.apache.spark.sql.Dataset` di piú alto livello rispetto a quella degli RDD, usata nel resto del processamento.

3.1 Preprocessamento

Quanto descritto è implementato nella classe *batch.App* in particolare nel metodo `preprocessing` e `load`. Le operazioni di preprocessamento sono state quelle di:

- eliminazione di tutte delle colonne non utili per le query: sono state mantenute `"tpep_dropoff_datetime"`, `"tip_amount"`, `"total_amount"`, `"tolls_amount"`, `"payment_type"`, `"tpep_pickup_datetime"`, `"PULocationID"`;
- eliminazione di tutte le righe contententi valori null
- eliminazione delle righe contenenti dati non appartenenti al periodo d'esame (sia per mese che per anno)
- conversione delle collonne nei tipi di dato attesi

3.2 Configurazioni injection

Pur avendo mantenuto la struttura delle query invariata, si e' sperimentato con due diversi approcci per l'injection dei dati.

3.2.1 Configurazione 1 . In questo caso, l'esito del preprocessamento non viene scritto su HDFS. Avendo già caricato i file parquet su HDFS, dopo aver definito la serie di trasformazioni che permette di ripulire il dataset dalle colonne non utili, dati mancanti o errati (con date non appartenenti al periodo specificato), il risultato del pre-processamento viene cachato ma non memorizzato su HDFS. La lettura dei dati necessari alle query avviene quindi in formato parquet. Le query lavorano sull'RDD ottenuto a seguito di una ulteriore conversione da `Dataset<Row>` a `JavaRDD<TaxiRoute>`.

3.2.2 Configurazione 2 . In questa seconda configurazione, il preprocessamento e' visto come totalmente separato dalla fase di processamento. Viene effettuata la stessa procedura di preprocessamento (pulizia e riduzione del dataset

alle sole colonne utili per le query) ma il risultato viene scritto su HDFS come file di testo, in formato CSV. Prima di iniziare il processamento, l'applicazione legge il file di testo contenente il dataset preprocessato e lo deserializza nelle corrispondenti istanze di `TaxiRoute` ottenendo mediante apposita map un `JavaRDD<TaxiRoute>` e a quel punto viene eseguito il processamento vero e proprio.

4 PROCESSAMENTO IN SPARK

4.1 Query 1

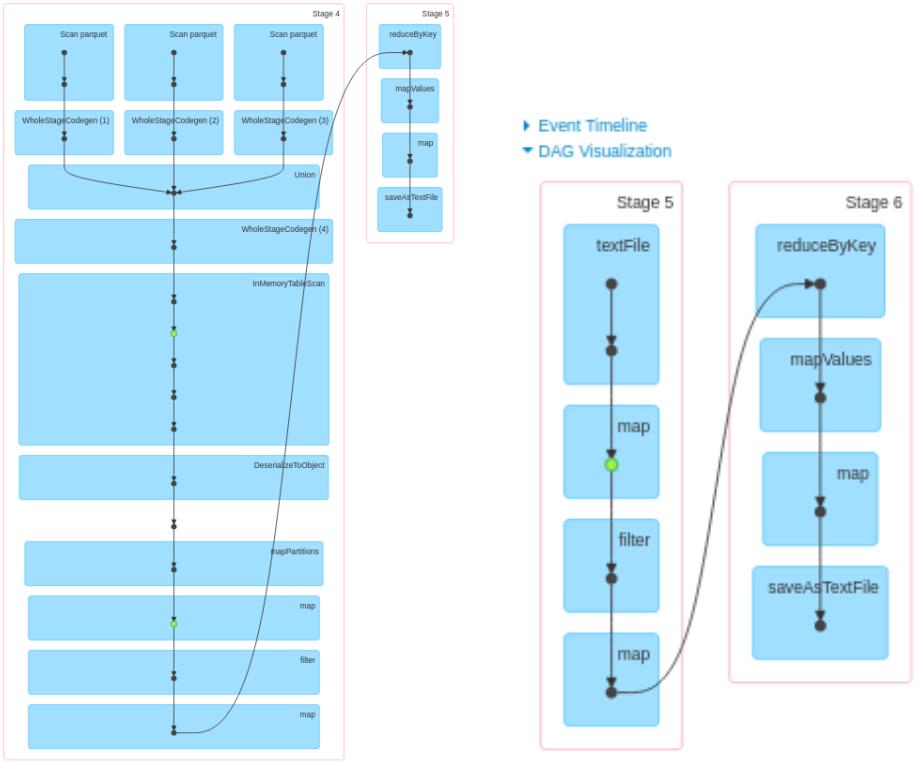

Fig. 2. Piano d'esecuzione visualizzato come DAG per la prima query. L'immagine a sx rappresenta il piano d'esecuzione nella Configurazione 1 : i dati letti dall'HDFS per effettuare la query sono in formato parquet. A sx si ha il piano di esecuzione nella Configurazione 2 in cui i dati di partenza sono file csv ottenuti dopo il preprocessing.)

La prima `map` dopo le trasformazioni sui dati (la sola `textFile` nella Configurazione 2 , la serie di trasformazioni più complesse sui file parquet) permette di ottenere un `JavaRDD<TaxiRoute>` che contiene tutte le colonne utili per il processamento.

Nella prima query il primo passaggio è una `filter` sul tipo di metodo di pagamento e per assicurarsi che il total pagato (`totalAmount`) sia diverso da 0. Il secondo controllo è necessario per non ottenere una divisione per 0 nei passaggi successivi che può essere evitata considerando queste come righe non rilevanti ai fini della query. Il filtro è la prima operazione effettuata per evitare ulteriori calcoli su quelle `TaxiRoute` che poi verrebbero comunque scartate.

Si effettua poi una `mapToPair` (nel DAG rappresentata come `map`) che definisce delle `Tuple2` per ottenere coppie chiave valore: la chiave è una stringa che rappresenta il mese, il valore una nuova tupla (`Tuple2<Double, Integer>`) nella quale si vuole tener traccia per ogni elemento su cui fare la media sia del suo contributo nella somma al numeratore sia della sua presenza nel conteggio per il denominatore. Il primo elemento della tupla é il rapporto `tip/(total -tolls)` per ciascuna riga, il secondo semplicemente un 1. La `mapToPair` causa il cambiamento di stage poiché vengono riorganizzate le partizioni secondo la chiave.

Con la `reduceByKey` si effettua la somma all'interno dello stesso mese sia degli addendi al numeratore (i rapporti calcolati nella map precedente) che degli 1 per il denominatore.

Infine si calcola mese per mese sulle partizioni già definite (con una `mapValues`) il valore del rapporto che fornisce la media richiesta.

4.2 Query 2

Il primo passaggio per la risoluzione di questa query è stata la creazione di un nuovo `JavaPairRDD` che contenesse le informazioni circa le fasce orarie in cui una certa corsa è stata effettuata. Si è definito un apposito metodo `.getAllHours` che indica, per ogni corsa, a partire dall'orario di partenza e da quello di arrivo, tutte le fasce orarie che la corsa ha toccato (quelle comprese tra i due orari). Ad ogni corsa è stata associato mediante `flatMap` un numero variabile di coppie chiave valore aventi per chiave una delle ore in cui la corsa é stata effettuata e per valore la corsa stessa. Il `JavaPairRDD` risultate è stato cachato percheé possa essere utilizzato da tutte e tre le sottoquery seguenti

4.2.1 Query 2.1: distribuzione oraria sulle zone delle corse. Si è effettuato un conteggio delle corse in una certa ora e in una certa zona mediante una prima `mapToPair` e conseguente `reduceByKey` : in particolare, la chiave della `mapToPair` è una stringa ottenuta come concatenazione di ora (chiave precedente) e zona. Associato a questa chiave si mette un 1 e si procede alla somma con una `reduceByKey` .

Si effettua una nuova `mapToPair` per suddividere i dati secondo la sola ora tenendo come valore una tupla contenente come primo valore la zona di partenza e come secondo la somma calcolata al passo precedente. Si raggruppa per chiave con una `groupByKey` per ottenere, data un'ora, il numero di corse che avevano come destinazione ognuna delle 265 possibili zone. La `groupByKey` è eseguita in quanto pemette di ottenere l'intera distribuzione ora per ora avendo inoltre, in questo caso, una garanzia forte sul numero di valori aggregati per chiave (al piú 265).

Si effettua infine una `mapValues` per ottenere, data una chiave una lista ordinata di valori indicizzata dalle zone e avente per entry il numero di corse per quella zona.

4.2.2 Query 2.2: media e deviazione standard della mancia. Si applica la stessa strategia della Query 1 per calcolare la media (si tiene traccia del numeratore, di un 1 per il denominatore mediante `mapToPair`). Per calcolare la deviazione standard si fa uso della formula:

$$var(X) = E[X^2] - (E[X])^2 \approx \frac{\sum_i x_i^2}{n} - (\frac{\sum_i x_i}{n})^2$$

che rende possibile calcolare i quadrati di ogni mancia (il numeratore del primo addendo) insieme al numeratore e al denominatore per la media, parallelizzando il calcolo anche per la deviazione standard. Con una `mapValues` si ottiene il valore di ciascun rapporto.

4.2.3 Query 2.3: il metodo di pagamento preferito. A partire dallo stesso rdd di partenza, con una `mapToPair` , se ne ottiene uno nuovo avente per chiave sia l'ora che il metodo di pagamento per ciascuna corsa e come valore un 1. Si

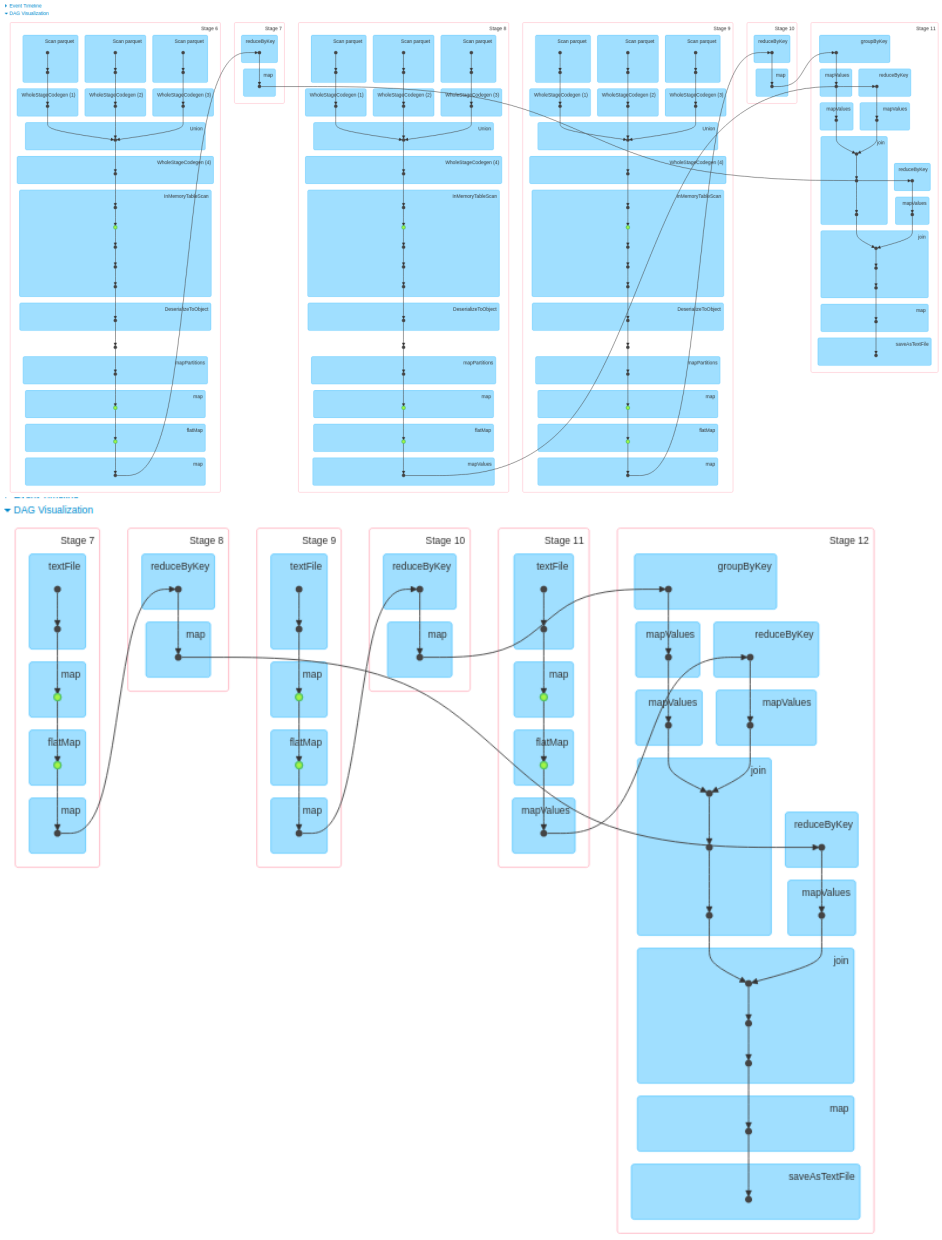

Fig. 3. Piano d'esecuzione visualizzato come DAG per la prima query. L'immagine in alto rappresenta il piano d'esecuzione nella Configurazione 1 : i dati letti dall'HDFS per effettuare la query sono in formato parquet. In basso si ha il piano di esecuzione nella Configurazione 2 in cui i dati di partenza sono file csv ottenuti dopo il preprocessing.)

effettua nuovamente una `reduceByKey` per ottenere il conteggio per ciascuna ora sul numero di volte in cui è stato scelto un certo metodo di pagamento.

Effettuato il conteggio si crea con un `mapToPair` un nuovo rdd avente per chiave la sola data e per valore una coppia contente il metodo di pagamento ed il conteggio del passo precedente.

Infine si sceglie il metodo di pagamento piú usato in una certa ora con una `reduceByKey` : per ogni coppia di tuple si sceglie solo la tupla con conteggio maggiore.

5 RISULTATI

Table 1. Tempi d'esecuzione

	Preprocessamento	Query 1	Query 2
Configurazione 1	-	$31386.4(\pm 2755)ms$	$32328.0(\pm 1039)ms$
Configurazione 2	$44620.6(\pm 3433)$	$10106.6(\pm 1192)ms$	$41632.4(\pm 7693)ms$

In tabella sono riportati i tempi d'esecuzione in entrambe le configurazioni: si è scelto di riportare entrambe le configurazioni per la sostanziale differenza di performance. Si pensa il drastico calo delle performance nella prima query nella prima configurazione rispetto alla seconda sia dovuto al formato di file usato per leggere l'input, file parquet, orientato alle colonne, e quindi poco adatto per una query su piú colonne come la query 1. Tuttavia nella query 2 si hanno performance migliori nella prima configurazione piuttosto che nella seconda. Si pensa che, anche in questo caso, la causa sia il formato di dati colonnare, adatto per tutte e tre le sottoquery.