**Esercizio 1**

Questo esercizio è stato svolto in Python, utilizzando le librerie Pandas e Numpy. In particolare, Pandas è stato utilizzato solo per leggere il file, dato che la sua funzione read\_csv risulta tra le più veloci per leggere file di alte cardinalità. Da qui in poi il programma può essere riassunto in due righe.

out = np.cumsum(np.sum(np.array\_split(data, 100), axis = 1))

Questa riga implementa 3 operazioni in catena:

* Innanzitutto, **np.array\_split** realizza il partizionamento dei dati in un numero di finestre popolate dallo stesso numero di dati. Tale operazione ritorna una lista di liste [100x100.000].
* Dopodiché, **np.sum** si occupa di applicare la somma di tutti gli elementi per ogni lista. In particolare, utilizzandola con axis = 1, l’operazione viene realizzata sui singoli array interni. In output ritorna un ndarray composto da 100 elementi.
* Infine, **np.cumsum** realizza la somma cumulata su tutti gli elementi degli array. In output ritorna di nuovo un ndarray composto da 100 elementi.

Arrivati a questo punto, l’output di questa riga completa il task richiesto dal test. Nella mia macchina locale questa operazione dura 0.85 sec. Però, visto che si richiede una lista indicizza e io a questo punto ritorno un ndarray, è stata applicata una riga di codice in più:

out = out.tolist()

Questa riga si occupa di trasformare l’ndarray in una lista. Ciò aggiunge circa 0.1 sec in più all’esecuzione del programma.

Nel caso in cui il file sia di dimensioni molto elevate si può rischiare di saturare, mentre non realizzerei cambiamenti a livelli di codice, dato che questo raggiunge l’obiettivo del task in modo molto pulito e facile da leggere, realizzerei il deploy dell’applicazione su un servizio scalabile come Dask per il calcolo parallelo.

**Esercizio 2**

Ho scelto di svolgere questo esercizio con Spark, data la tipologia di operazioni da realizzare sui dati. Infatti, il codice finale è molto diretto e facile da leggere e può essere riassunto in due righe di codice (dopo aver letto il file):

total\_avg = df.select(avg("Value")).collect()[0][0]

In questa riga di codice si realizza una semplice operazione di trasformazione dei dati tramite la funzione select. In generale, applichiamo la funzione avg (average) su tutta la colonna Value del dataset, che si occuperà di realizzare una media dell’intera colonna. Dopodiché tramite la funzione collect (in particolare collect()[0][0]) estraiamo il valore calcolato come float.

La seconda riga di codice è

gruppi = df.groupBy("ID").agg(avg("Value").alias("mean")).withColumn("Distance", col("mean")-total\_avg).orderBy("ID").show(truncate=False)

Qui si applicano una serie di operazioni a catena che ci permettono di ottenere l’output finale.

Per primo viene applicata un operazione di GroupBy, che ci permette di raggruppare i dati in base alla Colonna ID, e la quale è accoppiata all’operazione (agg(avg("Value").alias("mean"))). Per raggruppare bisogna indicare il tipo di aggregazione da realizzare sull’altra colonna, e in questo caso si è realizzata di nuovo una media dei valori, rinominando la colonna ottenuta a “mean“ tramite la funzione alias.

Dopodiché si applica una trasformazione dei dati utilizzando la funzione withColumn, la quale ci permette di aggiungere una nuova colonna come trasformazione di un’altra. In generale, costruiamo quindi tale colonna come la sottrazione del valore presente nella colonna “mean” meno il valore della media totale calcolato nella prima riga di codice.

Infine, è stato applicato un algoritmo di ordinamento basato sull’ID.

Nel caso in cui il file sia di dimensioni molto più elevate, i tempi di esecuzione vedrebbero sicuramente un aumento notevole, ma considerando che la soluzione è realizzata in Spark, non dovrebbero esserci grandi criticità.

Una possibile soluzione è quella di caricare il file sull’HDFS (Hadoop Distributed File System) o, se si opta per una soluzione in cloud, si può realizzare un deploy su cluster tramite il servizio EMR (Elastic Map Reduce) di AWS (Amazon Web Services).