Analisi Spark Core Covid-19 Opendata Vaccini

Roberto Pavia
Corso di laurea magistrale in Ingegneria Informatica
Università degli studi di Roma "Tor Vergata"
roberto.pavia@alumni.uniroma2.eu

***Sommario*—Questo documento descrive l'implementazione Spark Core per l'analisi di datasets opensource riguardanti le vaccinazioni Covid-19 in Italia. Vengono riportati in dettaglio i diversi frameworks utilizzati ed una visione dettagliata dell'architettura.**

I. Introduzione

Lo studio effettuato, risponde a due queries usando tre datasets differenti. Il primo dataset `punti-somministrazione-tipologia.csv` contiene i dati sui punti di somministrazione per ciascuna regione italiana. Questo dataset è composto da diversi campi e l'unico di interesse a quest'analisi è `nome_area`. Tipo di dato string che indica la denominazione standard dell'area dov'è presente tale hub vaccinale. Il secondo dataset `somministrazioni-vaccini-latest.csv` contiene dati sulle somministrazioni giornaliere dei vaccini suddivisi per regioni, fasce d'età e categorie di appartenenza dei soggetti vaccinati. Sono stati considerati i campi d'interesse `data_somministrazione`, `fascia_anagrafica`, `sesso_femminile` e `nome_area`. Il terzo dataset `somministrazioni-vaccini-summary-latest.csv` contiene dati sul totale delle somministrazioni giornaliere per regioni e categorie di appartenenza dei soggetti vaccinati. In questo caso i campi d'interesse sono `data_somministrazione`, `totale`, `nome_area`. Di seguito le motivazioni della scelta di questi campi e l'implementazione delle diverse queries.

II. Queries

A. Q1

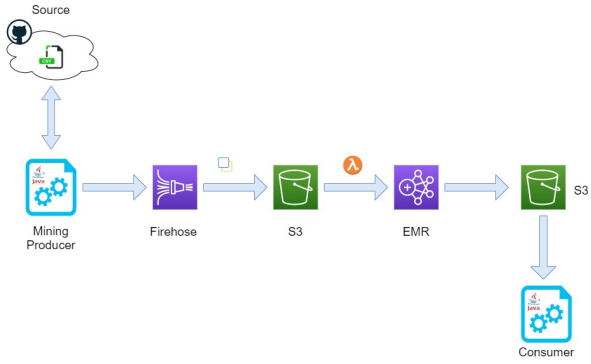
Nella prima query, per ogni mese solare e per ciascuna area si deve calcolare il numero medio di somministrazioni che è stato effettuato giornalmente in un centro vaccinale generico in quell'area e durante quel mese. Come da specifica, vanno considerati i dati a partire dal 01/01/2021. Tenendo quindi in considerazione i datasets `punti-somministrazione-tipologia.csv` e `somministrazioni-vaccini-summary-latest.csv` sono stati rispettivamente calcolati:

- Numbero di hubs vaccinali totali per ogni regione sfruttando il numero di occorrenze del dato `nome_area`
- Numero medio di vaccinazioni totali in un mese raggruppando il `totale` filtrato sulla `data_somministrazione` e mappato su `nome_area`

B. Q2

Nella seconda query, per le donne, per ogni fascia anagrafica e per ogni mese solare, bisogna determinare le prime 5 aree per le quali è previsto il maggior numero di vaccinazioni il primo giorno del mese successivo. Per determinare la classifica mensile e prevedere il numero di vaccinazioni si è dovuto considerare la retta di regressione che approssima l'andamento delle vaccinazioni giornaliere. Per ogni mese e categoria si è calcolata la classifica a partire dai dati raccolti dal 01/02/2021. Si è tenuto in considerazione il dataset `somministrazioni-vaccini-latest.csv`

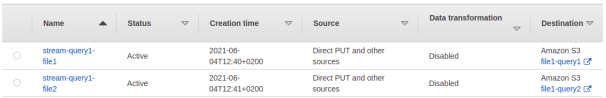
III. Architettura



A. Pulizia dei dati e Ingestion

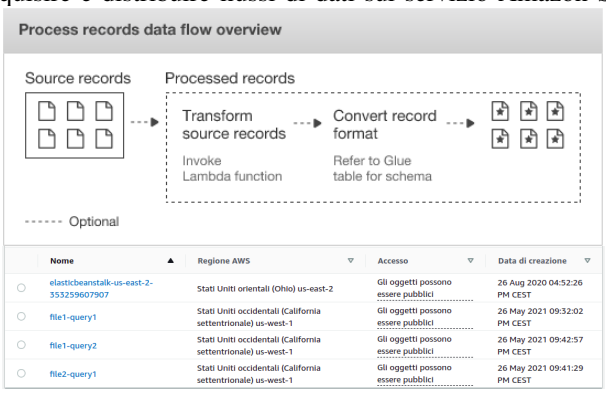
I datasets proposti sono in generale molto puliti, tuttavia è stato necessario prendere delle accortezze. Inoltre, come già notato, per rispondere alle diverse queries serve considerare un numero ridotto di colonne. Per sfruttare queste caratteristiche naturali dei dati si sarebbe potuto procedere in diversi modi:

- Parsing dei dati tramite un crawler Github con filtering and fix prima della fase di ingestion
- Filtering and fix durante la fase di ingestion
- Subito prima del processamento all'interno del cluster Spark

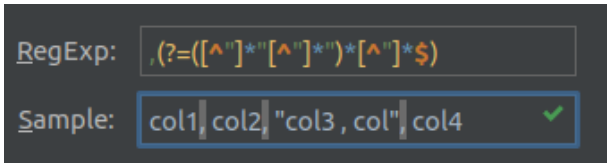


La scelta è stata quella di sfruttare un *Mining Producer* per parsare i dati dalla repository Github via HTTP request per poi sfruttare *Kinesis Firehose*. Trattasi di un servizio Amazon completamente gestito e di facile utilizzo per caricare flussi di dati in data lake, datastore e servizi di analisi in modo

affidabile. Ai fini di questo progetto è stato utilizzato per acquisire e distribuire flussi di dati sul servizio Amazon S3.



Attraverso l'invocazione di una funzione lambda sarebbe possibile convertire i records in ingresso all'agente Firehose in formato Parquet, oltre alla possibilità di lavorare sul numero di colonne e sul loro contenuto. Tutto questo è possibile farlo prima del caricamento riducendo al minimo lo storage utilizzato e migliorando anche gli aspetti di sicurezza. Non è stato possibile portare avanti questa soluzione in quanto si è deciso di lavorare unicamente con Spark Core puro. Lavorare su files in formato Parquet in Spark necessità dell'utilizzo di un Dataframe, istanziabile solo attraverso gli implicits di Spark SQL. Il preprocessing dei dati è stato quindi distribuito sul Mining Producer e sull'applicazione Spark che viene lanciata sul cluster EMR. In particolare, sul Producer, vengono ripuliti i datasets dall'header mentre nell'applicazione Spark si costruiscono gli RDD pulendo i dati tramite regexp. Un esempio di utilizzo della seguente regex:
(Toscana,"SPEDALI RIUNITI DI S, CHIARA",Territoriale)

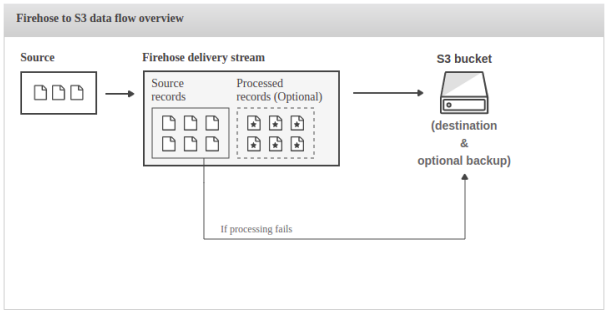


Non si esclude un futuro riutilizzo del servizio insieme ad un'applicazione Spark SQL in quanto Firehose è completamente gestito e permette il ridimensionamento automatico delle risorse in base al throughput dei dati, non richiedendo alcuna attività di amministrazione durante il funzionamento.

B. App Spark e storage

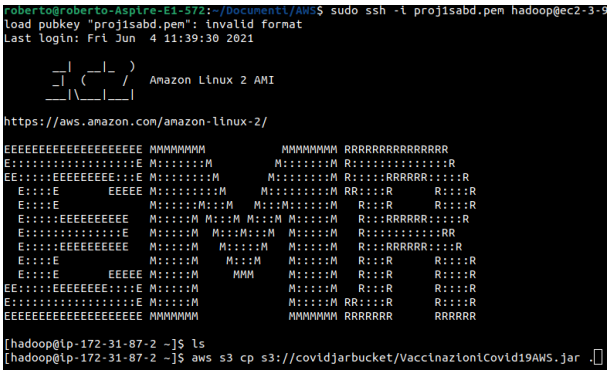
Per spostarci verso il servizio EMR dove avviene il vero e proprio processamento dei dati è necessario specificare una destinazione/bucket. Il modo più veloce e meno costoso di gestire il flusso di distribuzione è quello di sfruttare il servizio S3 di Amazon. I bucket S3 vengono indirizzati all'interno dei path relativi nell'applicazione Spark e da questi verranno letti i files/streams come RDD. Si è scelto di eseguire un cluster Spark 3.1.1 basato su Hadoop 3.2.1 YARN. Esistono due modi di procedere con la spark-submit:

- `spark-submit` via lambda function
- `spark-submit` manuale via SSH



Il submit via lambda function può avvenire grazie ad un trigger impostato su un preciso bucket S3. Per questo motivo si è deciso di procedere all'upload del dataset `somministrazioni-vaccini-latest.csv`, necessario per la seconda query, in modo separato dal Mining Producer. L'upload di questo su un bucket S3 avrebbe mandato un trigger alla "lambda-submit". Seguendo la documentazione ufficale di AWS Launch a Spark job si è tantato più volte di portare a termine il lavoro ma senza successo. Gli errori riscontrati sono dovuti ad un problema con le regioni AWS impostate sull'utilizzo della lambda function. Per proseguire comunque con il lavoro, si è deciso di simulare il triggering tramite l'upload manuale del datasets con il conseguente avvio del cluster Spark, prima tramite interfaccia grafica AWS, poi accedendo via SSH al master node per eseguire la submit. La configurazione del software prevede:

- Registrazione di log su bucket S3
- La modalità d'avvio è impostata su Cluster e non esecuzione fase (cosa che sarebbe stata prevista tramite lambda)
- La versione di EMR è la 6.3.0 per poter implementare il codice in Spark 3.1.1
- Il tipo di istanza utilizzata, con pattern general purpose, è una `m3.xlarge`. Un nodo master e due principali.



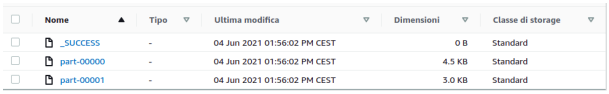
In parallelo all'avvio del CovidCluster si procede a generare il jar dell'applicazione Spark eseguendo una build artifacts del codice. Sarà necessario specificare la main class prima di fare la build del file jar da distribuire secondo lo schema SPMD. Nel caso in cui questo non venga fatto è comunque possibile specificare la main class tramite il comando di `spark-submit` usando il parametro `--class`. Dopo aver caricato il jar in S3, dalla console EMR acceduta via SSH copiamo in memoria il jar.

C. MLlib RDD-based

Per l'implementazione dell'algoritmo di regressione lineare si sono riscontrati problemi con l'utilizzo della classe `StreamingLinearRegressionWithSGD()` nella versione 3.1.1 di Spark. In particolare l'integrazione in Scala, seppur fatta seguendo la documentazione ufficiale è risultata problematica a livello di librerie. Viene infatti restituito un problema con l'inizializzazione del regressore. Sono state provate più versioni della stessa classe ma il risultato è stato sempre lo stesso anche in diversi ambienti di sviluppo. Si è deciso quindi di implementare l'algoritmo di regressione lineare classica direttamente in Scala nell'applicazione Spark. Nella classe `utils.Helper.scala` si può trovare il calcolo dei coefficenti della regressione mediante una soluzione veloce del calcolo della media. Questa soluzione prevede il calcolo della media `as we go`, ovvero aggiungendo il nuovo valore calcolato ogni volta alla media calcolata fino a quel punto, sfruttando la logica delle tuple.

D. Consumer, Monitoring e risultati

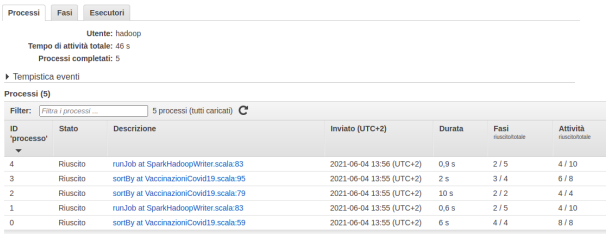
L'applicazione Consumer è speculare a quella del Producer. Allo stesso modo, sfruttando l'SDK di AWS disponibile sia per Kinesis Firehose che per S3, è stato simulato il download delle diverse partizioni create su un dato bucket dall'applicazione Spark. Le diverse partizioni vengono salvate all'interno di una cartella locale del Consumer, il quale potrà fare su queste monitoraggio, visualizzazione e statistiche varie. Non avendo la possibilità diretta in Spark Core di salvare un RDD come un file CSV, se non passando per il framework di SparkSQL, il consumer si occuperà di produrre i risultati da presentare lavorando sulle partizioni.



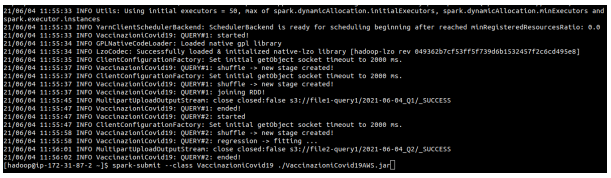
Come già spiegato in precedenza il cluster è composto da un nodo master e da due nodi core instance tutti su macchine m3.xlarge. Di seguito vengono riportati i relativi gruppi con le caratteristiche hardware specifiche.

La configurazione prevede una chiave di accesso al nodo master mostrato sopra per l'accesso SSH necessario ad eseguire `spark-submit`

Per concludere riportiamo le tempistiche prese direttamente dalla console di AWS. I diversi processi hanno terminato con stato di `Riuscito`. L'applicazione esegue le due query, con output di log minimale per evitare rallentamenti. Nei tempi riportati, oltre all'esecuzione delle diverse fasi, è incluso anche il tempo di salvataggio dei dati sui bucket.



L'applicazione Spark in output ritorna un messaggio di log per ogni nuovo stage creato nel DAG. Si è tenuto conto quindi di quando all'interno dello Spark Core avviene una fase di re-shuffle. Si può notare come la classe `MultipartUploadOutputStream` ritorna in output il momento di chiusura del canale verso i due bucket, sui quali scrivere le diverse parti dei risultati ottenuti. Come ultima riga il comando di `spark-submit` utilizzato con una jar prodotto senza la main class e quindi con la necessità di specifica come già spiegato in precedenza.



E. Conclusioni e limitazioni

Rispetto ai test fatti in locale, notiamo un uptime del cluster di circa il doppio sulla piattaforma di Cloud Computing AWS. Per un'applicazione molto semplice è difficile notare grandi differenze, anzi sembrerebbe funzionare meglio in locale. Mantenendo le configurazioni tra locale e cloud, il più simili possibile, in locale si sta lavorando con un `intel-core i5 4200u` equipaggiato con 8 GB di memoria RAM. Con applicazioni più complesse e query computazionalmente più pesanti il gap creatosi tra le prestazioni su cloud e quelle in locale tenderà a svanire invertendosi. Inoltre è facile prevedere tempi migliori in cloud senza l'aggiunta del framework di SparkSQL sopra Spark Core, perdendo però la possibilità di trattare i dati in modo agile e strutturato. In questo caso le dimensioni ridotte dei datasets, hanno influito sui tempi presentati. Un limite riscontrato è quello del `vendor lock-in`. Bisogna infatti notare come i servizi di AWS tendono ad integrarsi alla perfezione tra di loro ma si è limitati nell'integrazione con altri servizi di cloud computing come ad esempio quello offerto da Google.