

Sistemi e Architetture Big Data

Progetto 2 - A.A. 2020/2021

Studenti: Valerio Cristofori - 0300313

Tiziano Taglienti - 0304926

**Introduzione**

Lo scopo del progetto è rispondere a tre query su un dataset che riguarda dati provenienti da dispositivi *Automatic Identification System* (AIS) per scambiare informazioni riguardo il traffico di navi e garantire sicurezza in mare e nei porti.

Il file csv in esame contiene i dati geo-spaziali di interesse e viene processato seguendo la struttura dell’analisi in tempo reale.

Il progetto è stato sviluppato da un gruppo di due persone, in linguaggio Java e in ambiente Linux.

La piattaforma su cui è stato realizzato il progetto e su cui sono stati analizzate le metriche di latenza e throughput è un nodo standalone con sistema operativo basato su Linux, con 8 GB di RAM a disposizione e un processore AMD a 4.4GHz.

Per l’organizzazione del progetto è stata inizializzata una repository su GitHub.

(diciamo che abbiamo iniziato con storm? Forse nella relazione si può scrivere)

* Per processamento real time, scelto flink invece di storm perché:
  + Storm è un procesore di flusso di dati senza funzionalità batch.
  + Entrambi hanno l’obiettivo di elaborare flussi a bassa latenza mediante trasferimenti di dati in pipeline.
  + Flink offre un’API di livello più elevato rispetto a Storm – invece di implementare la funzionalità di un bolt con uno o più lettori e collezionisti, l’API Datastream di Flink fornisce funzioni come Map, GroupBy, Window e Join. Molte di queste funzionalità devono essere implementate manualmente quando si utilizza storm.
  + Differenza nell’elaborazione della semantica: storm garantisce elaborazione at least once, mentre flink è exactly once
  + Backpressure: lo streaming runtime di flink si comporta bene quando diversi operatori vanno a velocità diverse
  + User-defined state: flink permette ai programmi di mantenere uno stato personalizzato negli operatori. Può essere usato nel checkpointing per tolleranza ai guasti.
  + Streaming windows: il sistema di windowing di flink è potente e supporta molti tipi di finestre.

Oltre ad Apache Flink è stato utilizzato Kafka […]

– se mettiamo redis scrivere perché utilizzato redis.

**Architettura del sistema**

è stato usato un Docker compose per impostare i broker relativi a Kafka e per abilitare la connessione al server ZooKeeper.

Nel file *docker-compose.yml* si configura un ambiente il quale assicuri che il server ZooKeeper inizi sempre prima del server Kafka e si fermi dopo di esso. Nel nostro caso, il server ZooKeeper è in ascolto sulle porte 2181, 2888 e 3888 e sarà esposto sulle stesse porte per ogni client in esecuzione sull’host.

Inoltre, nello stesso ambiente, viene creato un cluster multi-node Kafka con tre nodi broker, assicurandosi che i container name siano univoci. Con la stessa logica usata per ZooKeeper, i *kafka\_broker* sono in ascolto sulla porta 9092.

– se mettiamo redis scrivere qualcosa su redis che sta nel docker-compose

è stata fatta questa scelta per […] (Facilitare sviluppo dell’applicazione in un ambiente locale? Focus sull’interazione tra i due framework? Forse 3 se aggiungiamo redis?).

Diverse fasi: acquisizione dati real time, trasformazione, analisi, come vanno in output i dati.

come salvo cose su Results

--- aggiungere screen dei DAG (o qua o per ogni query)

**Logica applicativa di Flink**

Lo scopo del progetto è quello di risolvere le tre query, analizzando i dati presi dall’AIS nel Mar Mediterraneo. Per lo svolgimento della logica applicativa, l’area marittima di interesse viene delimitata da una *Mappa* suddivisa in 10 settori per la latitudine e 40 settori per la longitudine.

Questa struttura aiuta a riconoscere le celle, tramite un indice univoco, nell’evoluzione delle query.

Prima di tutto, il dataset è stato ordinato sulla base del timestamp, come richiesto.

Prima query: cosa chiede, procedimenti, risultati finali

La prima query richiede di calcolare, per ogni cella del Mediterraneo Occidentale, il numero medio giornaliero di navi militari, navi per trasporto passeggeri, navi cargo e altre navi, calcolandolo in 7 giorni e in un mese di event time. Le navi vengono distinte in base al valore del loro SHIP\_TYPE.

Ai dati in ingresso viene applicata una funzione di *map* che restituisce solo le colonne di interesse dei record analizzati.

Inizialmente il flusso dei dati viene filtrato, per considerare solo le tuple che fanno riferimento a navi che si trovano nel Mediterraneo Occidentale, eseguendo un controllo sulla longitudine.

In secondo luogo si ottiene una *WindowedStream* attraverso delle trasformazioni di *keyBy* e *window* sui dati filtrati precedentemente. Questa operazione viene fatta distinguendo l’event time, utilizzando un *WeekWindowAssigner* o un *MonthWindowAssigner*.

Successivamente i dati vengono aggregati […] per poi essere mappati tramite una funzione che usa uno *StringBuilder* per costruire il risultato nella forma richiesta, il quale contiene il timestamp relativo all’inizio del periodo su cui è stata calcolata la media giornaliera, l’identificatore della cella e, per ogni tipo di nave, la media calcolata.

Il risultato finale è input di una funzione *addSink*, che richiama una funzione sink personalizzata (Flink ha dei connettori verso Kafka che sono implementati come funzioni sink). Kafka si integra con il meccanismo di checkpointing di Flink per offrire una semantica di processamento exactly-once.

Per la seconda query bisogna fornire una classifica delle tre celle più frequentate nelle fasce orarie 00:00-11:59 e 12:00-23:59, distinguendo i risultati per il Mediterraneo Orientale e Occidentale. Il grado di frequentazione indica il numero di navi diverse che attraversano la cella.

Dopo una prima analisi dei dati, si prendono gli identificatori dei viaggi per ogni cella e fascia oraria, tramite tumbling window. Poi i dati vengono aggregati […] e suddivisi in coppie dove la stringa corrisponde all’id della cella e l’intero si riferisce all’ora del giorno.

Viene di nuovo applicata una trasformazione *window* facendo una distinzione tra i due event time di una settimana e di un mese, per poi dare in pasto la *firstWindowedStream* a una funzione di reduce, con lo scopo di calcolare il grado di frequentazione totale di ogni cella per ogni fascia oraria.

In seguito viene suddiviso il flusso in base al tipo di mare, sempre grazie a un controllo sul valore della longitudine, per poi creare un’ultima *WindowedStream* e aggregare i dati […].

Infine i dati vengono mappati, come per la prima query, in modo da creare un output che contenga il timestamp e la classifica delle tre celle più frequentate per ogni fascia oraria.

Anche in questo caso al risultato viene applicata la funzione *addSink* personalizzata che chiama un Producer Kafka e garantisce la semantica exactly-once.

La terza e ultima query chiede di calcolare una classifica in tempo reale dei cinque viaggi con maggiore distanza percorsa. Questa viene calcolata aggiornando la distanza euclidea ogni volta che varia la posizione della nave in uno stesso viaggio.

Dopo aver creato una *WindowedStream* per intervalli di tempo pari a un’ora e due ore, i dati vengono aggregati sommando le distanze percorse da ciascun viaggio, ottenendo una stream di trip. Successivamente, per realizzare la classifica dei viaggi con distanza percorsa maggiore, si crea una *AllWindowedStream*, sempre tenendo in considerazione i due event time distinti. Ora i dati vengono aggregati [...Ranking Trip…] e si forma, come risultato, una stringa che contiene il timestamp e la classifica dei cinque viaggi con il punteggio di percorrenza maggiore.

[come viene calcolata la DISTANZA]

Come per le query precedenti, tramite la funzione *addSink*, Flink comunica con Kafka che gestisce il topic e garantisce una semantica exactly-once.

In ogni query misuro metriche (latenza e throughput) con REST API (sperando che funzioni)

**Considerazioni**

**Riferimenti**

REST API: <https://ci.apache.org/projects/flink/flink-docs-release-1.13/docs/ops/rest_api/>

https://ci.apache.org/projects/flink/flink-docs-release-1.13/docs/ops/metrics/#rest-api-integration