Sistemi e Architetture per Big Data

A.Y. 2020-2021

Analisi di dati marittimi geo-spaziali

Tiziana Mannucci 0285727
ACM Member
Master Degree Student
Università degli Studi di Roma Tor Vergata
Via del Politecnico 1
Roma, Lazio, Italia
Email:
tiziana.mannucci@alumni.uniroma2.eu

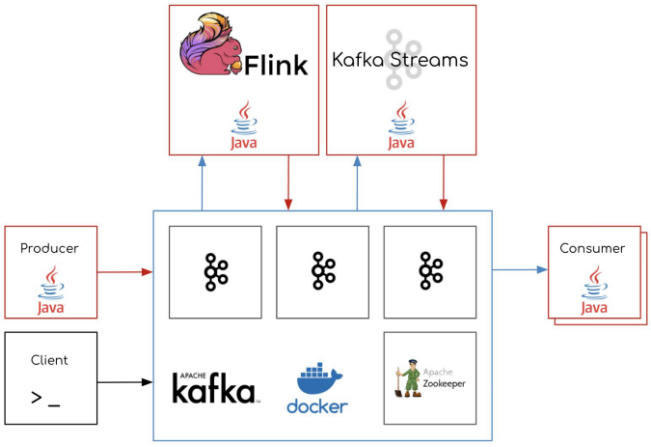
***Abstract*— In questo articolo viene argomentata l'analisi fatta su un dataset relativo a dati provenienti da dispositivi Automatic Identification System (AIS), andando a rispondere a delle query di interesse utilizzando un framework per il processamento streaming dei dati. Vengono illustrati infine i risultati dello studio ed un'analisi delle prestazioni.**

1 INTRODUZIONE

Nella presente relazione vengono esposte le soluzioni per rispondere a due query di interesse utilizzando Flink per il processamento streaming dei dati. È stato scelto quest'ultimo e non Storm in quanto risulta avere miglior throughput e permette di gestire il tempo in modi differenziati a seconda della tipologia di analisi.
Nello specifico si vuole effettuare lo studio in base all'Event Time, ovvero il tempo in cui l'evento si è effettivamente verificato.
Nelle sezioni 3 e 4 sono argomentate le soluzioni proposte rispettivamente per le query 1 e 2 e per ciascuna viene allegato il piano logico restituito da Flink.
Nella sezione 5 viene fatta l'analisi delle performance.

2 ARCHITETTURA DEL SISTEMA

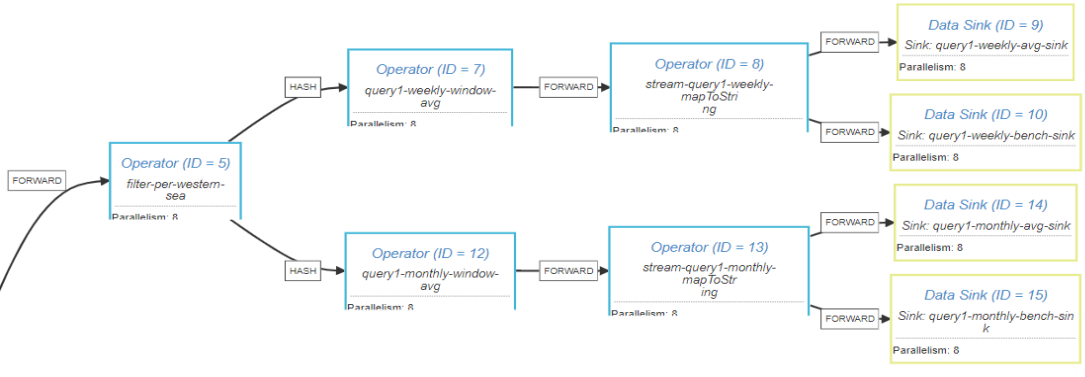
L'architettura utilizzata per l'analisi dei dati si compone di molteplici JVM (*Java Virtual Machine*), di quattro containers Docker e di un client Kafka su macchina locale.
Vengono utilizzati Docker containers per istanziare un cluster composto da tre Broker Kafka e da Zookeeper per il coordinamento distribuito.
Sulle JVM vengono eseguiti Producer, Consumers e l'applicazione DSP.
In locale viene eseguito un *client* Kafka per la creazione delle topiche necessarie al processamento.
Per simulare il flusso di dati a partire dal dataset fornito si usa un Producer che si occupa di leggere il file e fare il replay del contenuto su Kafka. Quest'ultimo è il framework che viene utilizzato per fare il *data ingestion*.
I dati sono veicolati dal Producer all'applicazione DSP che consuma i record dai Brokers, li processa in real-time, e pubblica i risultati su altre Kafka topiche di output che si differenziano in base alla query e alla finestra temporale di interesse.
Contestualmente ci sono tanti Consumers quanti sono le topiche di output e lavorano in parallelo consumando i record pubblicati dall'applicazione DSP scrivendoli su file in formato csv.



2.1 *Kafka*

Sono state create 5 topiche: una per le tuple
in *input* a Flink, due per l'*output* della prima *query* (settimanale e mensile) ed altre due per l'*output* della seconda *query* (settimanale e mensile).
Per incrementare la tolleranza ai guasti, ogni Kafka *topica* è impostata per avere un grado di replicazione pari a 2 (*leader* e *follower*) e, allo stesso tempo, una sola partizione. La scelta della singola partizione è dovuta alla necessità di mantenere le tuple ordinate all'interno del sistema di messaggistica poiché in Kafka l'ordinamento è garantito soltanto nell'ambito della singola partizione.

Figura 1: Query 1 Piano Logico



2.2 *Flink: Finestre Temporali User-Defined*

Una delle caratteristiche di Flink è la gestione flessibile del tempo e la possibilità di riferirsi ad esso in quanto *event-time*. Per la risoluzione delle query sono state implementate due finestre *MonthlyTumblingEventTimeWindow*, *WeeklyTumblingEventTimeWindow* che estendono la classe *TumblingEventTimeWindows* offerta da Flink.
Nello specifico sono state modificate le date di inizio e di fine delle finestre in modo tale che fossero allineate rispettivamente all'inizio (e fine) della settimana e del mese.
In questo modo le tuple con *event time* nella stessa settimana di un mese verranno indirizzate verso la medesima finestra settimanale e, similmente, tuple nello stesso mese, indipendentemente dal suo numero di giorni, verteranno sulla medesima finestra mensile.

2.3 *Flink : Watermarks*

Per gestire le tuple fuori ordine Flink sfrutta degli eventi di controllo che vengono iniettati nel flusso e che hanno associato un event time. Tuple ritardatarie, aventi timestamp successivo a quello associato al watermark, dovrebbero essere scartate a meno che non ci sia un ritardo ammesso.
Tale sistema va esplicitamente abilitato andando a specificare che l'event time corrisponde al campo timestamp delle tuple in ingresso estratte dal *flink-topic*. Tuttavia, questo sistema ha un effetto "collaterale" sulle finestre in quanto queste avanzano solo quando c'è un progresso di questi eventi di controllo. In un'applicazione reale in cui lo stream è *unbounded* questo è esattamente il comportamento che ci si aspetta, ma nel nostro caso il dataset fornito ha una dimensione limitata e non verrebbe processato nella sua interezza proprio perché, in assenza di ulteriori dati in ingresso, le finestre non avanzerebbero né produrrebbero risultati.

Per risolvere il problema è stato generato un *fake_dataset* sulla base di quello reale e che rappresenta il naturale avanzamento del flusso e, di conseguenza, anche quello delle finestre.

3 QUERY 1

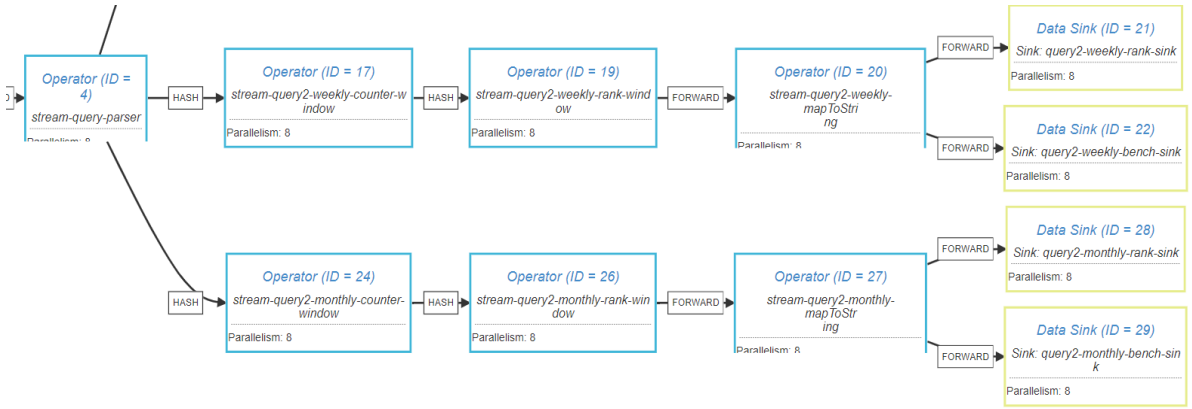
3.1 *Topologia*

È richiesto di calcolare la media giornaliera delle navi in ciascuna cella del Mar Mediterraneo Occidentale e per ciascuna tipologia di nave. Il calcolo deve essere effettuato su due finestre temporali: settimanale e mensile.

In Figura 1 è rappresentato una parte del DAG che riguarda la query 1. Le tuple vengono filtrate per mantenere solo quelle che riguardano il Mar Mediterraneo Occidentale.
Il medesimo flusso di dati viene inviato ad entrambe le finestre temporali (settimanale e mensile).
La trattazione che segue fa riferimento solo ad una delle due, in particolare quella settimanale, poiché vengono effettuate le medesime operazioni per l'altra.

Il flusso viene ripartito tra le finestre (settimanali) in base all'identificativo della cella usando una *keyBye e* per ciascuna tipologia di nave viene effettuata la media giornaliera.
Nello specifico la media viene calcolata sui giorni effettivamente osservati dalla finestra e non sul totale dei giorni settimanali altrimenti si avrebbe una media sottostimata.

Figura 2: Query 2 Piano Logico



3.2 *ProcessWindowFunction with Incremental Aggregation*[1]

Per fare la media si usa una *ProcessWindowFunction* che viene combinata con un'*AggregateFunction* per aggregare gli elementi in modo incrementale e progressivamente con il loro arrivo alla finestra. Quando questa viene chiusa la *ProcessWindowFunction* verrà fornita con il risultato aggregato. Tale metodo è più efficiente dell'utilizzo della sola *ProcessWindowFunction* per fare una semplice operazione di aggregazione e permette ugualmente di disporre delle meta-informazioni addizionali che si hanno con essa.

3.3 *Sink*

I valori aggregati e prodotti dalle finestre vengono mappati in stringhe ed indirizzati ai nodi Sink. Nello specifico ci sono due Sink per ciascuna *finestra logica temporale*. Due Sink si occupano di pubblicare i risultati sulle topiche Kafka opportune e gli altri due servono per valutare le performance, in particolare per avere una stima sperimentale del throughput medio e della latenza.

4 QUERY 2

4.1 *Topologia*

Nella seconda query è richiesto di calcolare le tre celle più frequentate nel Mar Mediterraneo Occidentale e le tre più frequentate del Mar Mediterraneo Orientale, per ciascuna delle due fasce orarie: 00:00- 11:59 e 12:00-23:59.
Similmente alla prima si richiede di rispondere considerando le due finestre temporali: settimanale e mensile.

Per rispondere alla query si usano le stesse finestre user-defined descritte in sezione 2.2.

La trattazione che segue fa riferimento solo ad una delle due finestre temporali, in particolare quella settimanale, poiché vengono effettuate le medesime operazioni per l'altra.

La computazione della classifica viene suddivisa in due fasi:

1. calcolo del totale dei tripId differenti per cella e fascia oraria.
2. calcolo della classifica per Mare e fascia oraria basato sui valori di 1.

4.1.1 *Fase 1*

Le tuple vengono ripartite in base all'identificativo della cella ed in base al Mare (Occidentale o Orientale). Quest'ultima informazione è ridondante in quanto è implicita nell'identificativo della cella, ma viene ugualmente specificata per poter essere recuperata velocemente nella Fase 2.
Anche in questo caso si usano finestre con il pattern visto in sezione 3.2 (*ProcessWindowFunction with Incremental Aggregation).*
La funzione di Aggregazione ha lo scopo di calcolare il numero di tripId differenti osservati per ciascuna fascia oraria. A tal fine si usano due oggetti di tipo *HashSet* (uno per fascia oraria) che permettono di aggiungere tripId senza che ci siano dei duplicati.
Vengono restituiti degli oggetti di tipo *SeaCellOutcome* che contengono la taglia dei Set, ovvero il totale dei tripId diversi osservati e altre meta-informazioni tra cui la tipologia di Mare.

[1] https://ci.apache.org/projects/flink/flink-docs-release-1.13/docs/dev/datastream/operators/windows/#processwindowfunction-with-incremental-aggregation

4.1.2 *Fase 2*

In Figura 2 si nota che il flusso in output dalla prima finestra (logica) viene indirizzato verso una seconda finestra che ha lo scopo di stilare la classifica. Il flusso in uscita dalla Fase 1 viene ripartito solo in base al Mare (Occidentale o Orientale).
Anche in questo caso si usa il pattern visto in sezione 3.2 (*ProcessWindowFunction with Incremental Aggregation).*
La funzione di Aggregazione ha lo scopo di collezionare in due liste (una per ciascuna fascia oraria) le informazioni sull'identificativo della cella ed il totale dei tripId che sono ad essa associati. Tali informazioni sono contenute in *SeaCellOutcome* e calcolate nella Fase 1.
Per stilare la classifica si ordinano in modo decrescente le due liste e si prendono i primi tre identificativi delle celle con più alto grado di frequentazione.
Al completamento della finestra le due classifiche vengono restituite in output dalla Fase 2.

4.1.3 *Sink*

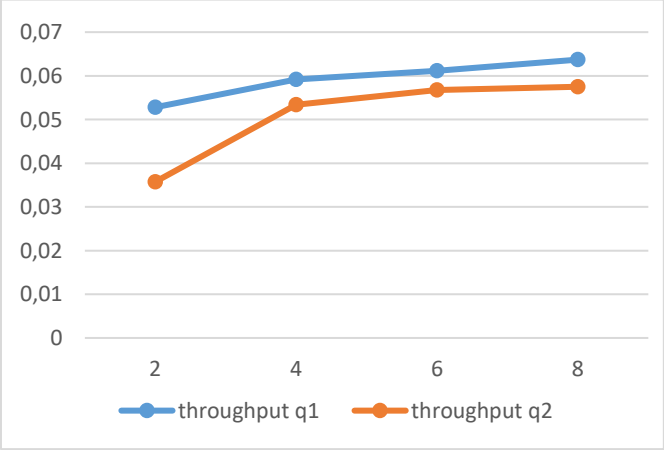
Vengono usati due nodi Sink per la pubblicazione dei risultati su due Kafka topiche, una per finestra temporale, ed altre due per valutare le performance, in particolare per avere una stima sperimentale del throughput medio e della latenza.

5 ANALISI PERFOMANCE

Tabella 1: Throughput Medio Weekly

Parallelismo	P=2	P=4	P=6	P=8
Query	Tuple/ms	Tuple/ms	Tuple/ms	Tuple/ms
1	0,05281	0,05919	0,06117	0,06371
2	0,03575	0,05342	0,05677	0,05750

Figura 3: Grafico Throughput medio al variare del grado di parallelismo

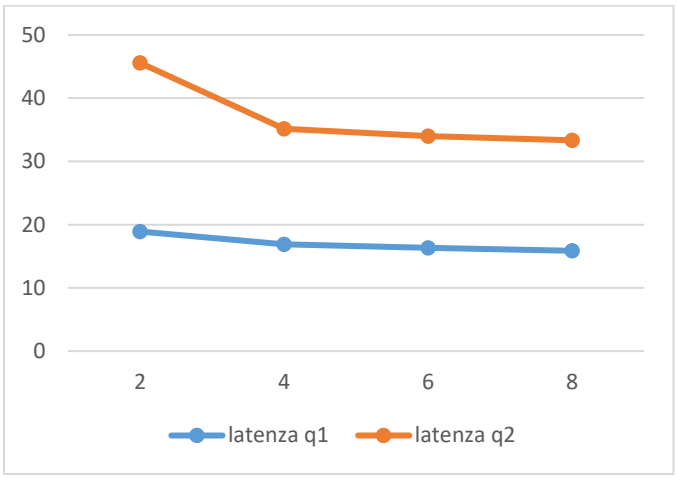


L'applicazione è stata eseguita più volte facendo variare il grado di parallelismo di tutti i nodi ad eccezione di quelli Sink per il benchamrking che sono stati forzati ad 1 per avere delle statistiche coerenti. Nello specifico sono state effettuate 10 esecuzioni per ciascun grado di parallelismo ed è stata fatta la media dei dati prodotti dai Sink per il throughput e la latenza.
In Tabella 1 sono riportati i valori medi del throughput e sono visualizzati in Figura 3. Nello specifico si evidenzia l'andamento crescente del throughput all'aumentare del grado di parallelismo[2] per entrambe le query.
Tale andamento è naturale in quanto più operatori lavorano in parallelo ed il flusso viene ripartito fra di essi con il conseguente aumento del numero di record prodotti in output per unità di tempo.
In Figura 4 viene riportato l'andamento delle latenze che è inversamente proporzionale al throughput (*"lower is better").*

Tabella 2: Latenza Media Weekly

Parallelismo	P=2	P=4	P=6	P=8
Query	ms	ms	ms	ms
1	18,912	16,895	16,342	15,866
2	26,67	18,25	17,659	17,473

Figura 4: Grafico Latenza media al variare del grado di parallelismo



5.1 *Architettura per il test*

- CPU: Intel(R) Core(TM) i7-9700K CPU @ 3.60GHz sbloccato (up to 4900MHz).
- Motherboard: MPG Z390 GAMING PRO CARBON AC.
- RAM: 16GB DIMM DDR4 Synchronous 3200 MHz (0.3 ns) CMK16GX4M2B3200C16 a 64bit.

[2] Il massimo grado utilizzato è stato "8" pari al numero di core del processore della macchina su cui sono stati eseguiti i test.