Analisi di dati marittimi geo-spaziali con Flink

Cecilia Calavaro
Corso di laurea magistrale
in Ingegneria Informatica,
Università degli studi di Roma
"Tor Vergata"
cecilia.calavaro@alumni.uniroma2.eu

***Abstract*—Sistema che risponde a delle queries riguardantiun dataset relativo a dati marittimi geo-spaziali provienienti da dispositivi AIS, utilizzando il framework di processamento Apache Flink.**

1. Architettura di Sistema

E' stata realizzata un'applicazione per rispondere ad alcune queries riguardanti l'analisi di dati marittimi geo-spaziali provenienti da dispositivi Automatic Identification System (AIS), utilizzando il framework di data processing **Apache Flink**. In particolare, il processamento dei dati è di tipo stream, quindi riguarda flussi di dati che arrivano al sistema in tempo reale.
Le queries a cui si è risposto utilizzando Apache Flink sono la 1, la 2 e la 3. La piattaforma utilizzata per il processamento delle queries è un nodo standalone su cui sono stati avviati diversi cluster: uno Flink per il processamento, un cluster **Kafka**, che è un sistema a coda di messaggi distribuito realizzato pe rendere lo streaming dei dati disponibile a molteplici consumers, ed infine un cluster Zookeeper, un servizio centralizzato necessario per la configurazione e sincronizzazione di servizi distribuiti, in questo caso per il funzionamento di Kafka. Per tale configurazione è stato utilizzato **Docker Compose**, utile a definire ed eseguire applicazioni multi-container di tipo Docker. Tramite un file YAML, infatti, sono stati configurati i servizi dell'applicativo, che consistono in un'immagine che definisce il container per il master di flink, una per i diversi worker, una per zookeeper ed infine l'immagine del kafka broker.

2. Scelte del design d'implementazione

L'applicazione è stata scritta nel linguaggio di programmazione Java ed è stata gestita con Apache **Maven**, in modo da facilitare il download automatico delle librerie necessarie al progetto e la risoluzione di eventuali dipendenze. In questo modo si è potuta raggiungere una facile portabilità del codice applicativo, sopratutto in termini di configurazione dell'ambiente. L'altro vantaggio nell'utilizzo di Maven consiste nella creazione di una **jar** del progetto che possa contenere tutte le dipendenze necessarie per compiere il submit dell'applicazione.
Si è deciso di utilizzare **Kafka** per simulare lo stream processing perché permette la comunicazione fra diverse componenti software che producono e consumano flussi di dati per scopi diversi. La più importante astrazione in Kafka è il concetto di **topic**, che è gestito come uno stream logico di dati che consiste in diverse partizioni. Il servizio che inserisce dati nel topic è chiamato producer, mentre quello che ne legge il contenuto è detto consumer. In quest'applicazione sono state realizzate due applicazioni, rispettivamente **MyProducer** e **MyConsumer**, per cercare di ricreare nel modo più fedele possibile uno scenario reale. Una volta che il producer ha letto i dati dal dataset originario in formato csv, invia ogni riga al broker di Kafka, simulandone, come in un contesto reale, un ritardo (che può opportunamente essere accelerato o meno all'interno del codice sorgente). D'altra parte il consumer, in ascolto, riceve i dati e li processa in real time, secondo quanto richiesto da specifica di progetto. Il dataset inoltre è stato ordinato all'inizio dello svolgimento del progetto utilizzando Excel, quindi l'elaborazione delle query avviene su dei dati nella corretta sequenza temporale.

Facendo invece riferimento alla struttura della repository GitHub del progetto ("ceciliacal/sabd_project2"), il codice sorgente si trova nella cartella **"src/main/java"** ed è stato suddiviso nei seguenti packages: "**connectionToKafka**", contenente le classi "**MyProducer**" e "**MyConsumer**", "**query1**", "**query2**", "**query3**" ed infine "**utils**", avente delle classi di supporto all'elaborazione degli outputs dell'applicazione.

D'altra parte, i files necessari al deployment dell'applicazione si trovano nella cartella "**docker**", che contiene il file **docker-compose.yml** ed un **Dockerfile** per il setup dell'applicazione del producer. Si è scelto di utilizzare docker compose perché rappresenta una soluzione sia flessibile per quanto riguarda la scalabilità dell'applicazione, sia molto pratica grazie alla semplicità della configurazione del suo ambiente, poiché basta specificare ogni servizio necessario all'interno di un unico file. Il suo utilizzo infatti permette di configurare e avviare molteplici containers Docker sullo stesso host senza eseguirli separatamente. Inoltre, nonostante la piattaforma consista in un nodo standalone, l'approccio multi-container permette di simulare uno scenario di un sistema che lavori con i Big Data più realistico, in cui il processamento venga suddiviso fra più macchine, che in questo caso sono rappresentate appunto da molteplici containers.

3. Descrizione dell'implementazione

Il codice sorgente per rispondere alle query 1, 2 e 3 è stato suddiviso in diversi package, che sono descritti di seguito:

3.1. Package "connectionToKafka"

Le classi contenute in questo package sono:

- **MyProducer**: è la classe che realizza il Producer Kafka, il quale si occupa di leggere ed inviare i dati ad un certo topic del Broker. A valle della creazione del producer, è

necessario specificarne le proprietà che lo definiscono. Come si può vedere nel metodo **createProducer()**, sono richieste in particolare quattro diverse configurazioni: un kafka server, per specificare l'host name e la porta del broker kafka; un identificativo univoco del producer, che rappresenta il kafka client; ed infine i Key Value Serializers, che permettono di definire come gli oggetti che vengono inviati sono tradotti rispetto al formato byte-stream usato da Kafka. In questo caso, poiché sia la key che il value dei messaggi sono di tipo Strings, viene utilizzata la classe **StringSerializer** fornita da Kafka.
All'interno di questa classe comunque è contenuto il metodo **main()**, che invoca a sua volta un metodo chiamato **publishMessages()**. Quest'ultimo contiene tutto il flusso logico del producer e ha lo scopo, appunto, di pubblicare messaggi, andando a leggere riga per riga il file csv contenente il dataset di input. A ogni riga, viene effettuato un parse dei vari campi contenuti in essa e viene prelevato quello contenente il timestamp (event time). Ogni timestamp viene aggiunto ad una lista che conserva tutti i timestamps precedenti rispetto alla riga corrente. Questo permette di poter effettuare il **replay** dei dati, andandoli ad inviare con un ritardo più o meno variabile a seconda della differenza tra il timestamp corrente e quello precedente. Viene invocato quindi il metodo **simulateStream()** che calcola il tempo di sleep dell'applicazione fra l'invio di due messaggi consecutivi, che andando a guardare al valore della differenza in minuti fra due i timestamps restituisce lo stesso valore, ma con i millisecondi come unità di tempo. Lo **sleep time** del processo fra l'invio di due messaggi consecutivi è dato quindi da tale differenza moltiplicata che viene moltiplicata per un fattore di accelerazione, in modo da poter inviare i dati più lentamente o più velocemente a seconda del valore che gli si assegna. Una volta che simulateStream() ha finito il suo calcolo, il flusso d'esecuzione torna nel metodo publishMessages() e vengono inviati i messaggi al broker di kafka al topic chiamato " **source**".

- **MyConsumer**: questa classe implementa l'applicazione che legge i messaggi che vengono pubblicati su un certo topic. Di conseguenza, nonostante il progetto sia unico, essa rappresenta un'applicazione a se stante, infatti contiene un altro **main** method. All'interno di tale metodo viene innanzitutto invocata la routine **createConsumer()** in cui viene appunto creato il consumer e, analogamente alla creazione del producer, si occupa di specificare prima le proprietà necessarie alla sua instanziazione: il kafka server, cioè host name e porta del broker contenente il topic di cui si vogliono consumare i messaggi, l'identificatore del consumer group, l'offset da cui leggere i messaggi, ed infine i Key Value Serializers. Una volta definite queste configurazioni, viene creato un **FlinkKafkaConsumer** che legge il topic "source" (creato prima dal producer). Una volta restituito il consumer, nel **main()** vengono definiti i **watermarks** aventi durata pari ad un minuto, e poi invocato un ulteriore metodo chiamato **createEnviroment()** il quale ha l'obbiettivo di creare l'ambiente di esecuzione. Dall'enviroment viene quindi estrapolata la sorgente, che collega flink a kafka, da cui si prelevano i dati di input. Flink lavora con la classe DataStream, quindi viene creato un DataStream di tipo "**Ship**". Quest'ultimo oggetto è un'istanza della classe entity **Ship**, definita all'interno del progetto nel package "utils" e che rappresenta il messaggio con cui si sta lavorando: ha diversi attributi, prelevati sia dai vari campi delle righe del dataset, sia creati appositamente per poter manipolare più facilmente i dati durante il processamento delle query. Questo datastream di oggetti "Ship" quindi viene creato utilizzando due operatori di flink: una **map** e una **filter**, che hanno rispettivamente il compito di andare a suddividere i campi di ogni riga e di creare un oggetto Ship passandoli nel costruttore di tale classe, e di filtrare i dati considerando solo quelli contenuti nelle celle definite nella specifica del progetto. Per quanto riguarda la creazione dell'oggetto all'interno della map, nel momento dell'invocazione del costruttore viene calcolato, per ogni istanza, anche la cella a cui appartiene, il tipo di mare d'appartenenza (cfr le specifiche progettuali) e viene identificata la tipologia di nave. Infine, vengono processate in sequenza le tre query, andando a invocare i metodi **runQuery1**, **runQuery2** e **runQuery3**. Ognuno di essi prende come parametro d'ingresso un DataStream "Ship", in modo da avviare immediatamente la computazione per la risoluzione delle queries.

3.2. Package "query1"

In questo package sono contenute tutte le classi necessarie per la risoluzione della query1. La classe che contiene il flusso logico è " **Query1.java**", che quindi verrà illustrata di seguito.

La classe Query1 ha un solo metodo che si chiama **runQuery1**, il quale riceve in ingresso il DataStream. Sul flusso di dati viene applicato un'operatore **filter**, per prendere solo gli oggetti la cui cella sia collocata nel Mar Mediterraneo Occidentale, ed una **keyBy** per partizionare il flusso di dati in base alla chiave specificata, in questo caso la cella d'appartenenza ad un certo event time.
Le finestre utilizzate tramite l'operatore **window** sono entrambe di tipo tumbling e basate su event time. Viene quindi risolta la query prima su una finestra temporale con durata di 7 giorni. Sul flusso di dati, pertanto, è invocato l'operatore "window": ciò comporta che gli operatori successivi andranno a processare solo le tuple che in quel momento sono presenti nella finestra. Viene poi chiamato l'operatore **aggregate**, a cui vengono passati come parametri d'ingresso i costruttori di due classi, " **AverageAggregate**" e " **Query1ProcessWindowFunction**".

- Classe **AverageAggregate**: questa classe implementa la classe "AggregateFunction" di Flink. Prende in ingresso un oggetto di tipo Ship, come accumulatore un oggetto di tipo **AccumulatorQuery1**,e in output un oggetto **OutputQuery1**, entrambi creati appositamente. AverageAggregate implementa quindi i 4 metodi previsti da Flink "createAccumulator", "add", "getResult" e "merge", e il suo funzionamento è strettamente correlato con la struttura dell'Accumulator. La classe AccumulatorQuery1 ha un attributo Map<String, List<String» chiamato "**countShipType**" che quindi ha come chiave una stringa che identifica il tipo di nave e come valore la lista degli ID delle navi che in quei 7 giorni sono passate in quella determinata cella. Una singola nave viene però contata in una cella soltanto una volta al giorno: alla lista

di stringhe di countShipType viene aggiunto l'id della nave concatenato con il giorno del timestamp di quel dato. La classe AccumulatorQuery1 inoltre ha un metodo chiamato " **addShipPerType**()", che aggiunge un nuovo id alla lista di un certo tipo di nave, soltanto se esso non è già presente, in modo da contare una specifica nave una sola volta all'interno di una giornata e in quella determinata cella, ma in un giorno diverso sarà invece aggiunta e quindi contata nuovamente.
Il metodo " **addShipPerType**()" in AccumulatorQuery1 viene chiamato dal metodo "add" di AverageAggregate ogni volta che il sistema vede una nuova tupla, e, una volta che tutte quelle contenute nella finestra sono state aggiunte, viene finalmente invocato il metodo "getResult". All'interno di esso viene chiamato il costruttore della classe OutputQuery1, passandogli come parametri l'attributo " **getCountShipType**" dell'accumulator, il quale ormai conterrà una mappa con 4 elementi diversi: uno per ogni diverso tipo di nave con la propria lista di ship id.
La classe OutputQuery1 ha come attributi l'id della cella corrente, ossia quella definita dalla keyBy iniziale, una data, che in seguito verrà associata al timestamp relativo all'inizio del periodo su cui è calcolata la media, ed una mappa Map<String, Integer> chiamata "countType" che viene popolata all'interno del costruttore OutputQuery1. La chiave della mappa, di tipo String, rappresenta il tipo di nave, e il value, ossia l'intero, rappresenta la dimensione della lista associata a quel tipo di nave (presa dall'attributo getCountShipType dell'accumulator che gli è passato come parametro del costruttore).

- Classe **Query1ProcessWindowFunction**: il costruttore di questa classe è il secondo parametro dell'operatore aggregate, ed estende la classe di Flink "ProcessWindowFunction". Il suo obbiettivo è quello di andare a settare i rimanenti attributi delle diverse istanze della classe OutputQuery1, cioè la data del timestamp dell'inizio della finestra, l'id della cella che è la chiave della partizione momentaneamente processata, e di resituire una collection di oggetti di tipo OutputQuery1.

Tornando al metodo runQuery1 della classe Query1, dove vengono chiamati i diversi operatori, dopo la aggregate (appena illustrata) viene chiamato un operatore **map**, che tramite una lamda function invoca il metodo "**writeQuery1Result**()" appartenente alla classe OutputQuery1. Quest'ultimo si occupa appunto di scrivere il risultato della query, andando a calcolare la media giornaliera: si divide il numero di navi di ogni tipo per il numero di giorni della tumbling window (quindi in questo caso 7, successivamente 28). Esso restituisce una stringa contenente l'output della query1 che verrà poi passata al **Sink** verso Kafka, ed inoltre crea anche il file **csv** contenente i risultati all'interno della cartella "results".
Infine, dopo la map, viene chiamato l'operatore **addSink** in cui viene creato un nuovo KafkaProducer che invia i risultati verso un nuovo topic di Kafka, chiamato "QUERY1". D'altra parte, la computazione sulla finestra temporale di un mese è identica a quella di 7 giorni, con la sola differenza che la durata della finestra è di 28 giorni (4 settimane).

Una nota importante è che il flusso logico che c'è dietro alla risoluzione della query1 è lo stesso anche nella 2 e nella 3, ossia nell'utilizzo di operatori window, aggregate, map e addSink. Le differenze sostanziali sono nelle classi che implementano l'Accumulator nelle diverse query.

3.3. Package "query2"

In questo package sono contenute tutte le classi necessarie per la risoluzione della query2. La classe che contiene il flusso logico è " **Query2.java**", che quindi verrà illustrata di seguito.

La classe Query2 ha un solo metodo che si chiama **runQuery2**(), il quale riceve in ingresso il DataStream. Sul flusso di dati viene chiamato l'operatore **keyBy**, che partiziona il flusso di dati per tipo di mare (quindi Mar Mediterraneo Occidentale oppure Orientale). Viene poi applicata una finestra anche qui di tipo tumbling basata su event time di dimensione pari a 7 giorni, ed in seguito, l'operatore **aggregate**, che riceve come parametri d'ingresso i costruttori delle classi **RankAggregate** e **Query2ProcessWindowFunction**.

- Classe **RankAggregate**: questa classe implementa la classe "AggregateFunction" di Flink. Prende in ingresso un oggetto di tipo Ship, come accumulatore un oggetto di tipo **AccumulatorQuery2** e in output un oggetto **OutputQuery2** (entrambi creati appositamente). Come spiegato nel paragrafo 3.2 ("class Average Aggregate"), il suo funzionamento è strettamente correlato con la struttura dell'Accumulator. Per fornire la classifica delle tre celle più frequentate nelle due fasce orarie di servizio 00.00-11.59 e 12.00-23.59, la classe AccumulatorQuery2 ha due attributi di tipo Map<String, List<String» chiamati "**am**" e "**pm**", per distinguere la fascia oraria mattutina da quella pomeridiana. La chiave, ossia la stringa, rappresenta la cella mentre il valore rappresenta la lista di id delle navi presenti in quella cella. Inoltre, ha due metodi chiamati "**addAM**()" e "**addPM**()" per aggiungere l'id di una nuova nave alla lista di "am" o di "pm". Poiché nella specifica progettuale viene indicato che il grado di frequentazione di una cella è il numero di navi **DIVERSE** che attraversano la cella nella fascia oraria in esame, in questo caso una nave viene considerata solamente una volta, nell'arco di 7 giorni, per ogni cella distinta. A seconda se il timestamp dell'oggetto Ship abbia un orario che rientri nella fascia della mattina o del pomeriggio, viene quindi chiamato o il metodo "addAm" o "addPm" all'arrivo di una nuova tupla nella finestra.
Una volta che il sistema passa all'invocazione del metodo "**getResult**()", s'istanzia un nuovo oggetto di tipo **OutputQuery2** chiamando il costruttore di tale classe, che riceve come parametri d'ingresso le due mappe "am" e "pm" dell'accumulator.
La classe OutputQuery2 serve per calcolare la classifica vera e propria: ha due attributi "amRank" e "pmRank" che sono due liste List<Tuple2<List<String>, Integer» , e che contengono delle tuple aventi come primo valore una lista di celle, e come secondo valore un intero, il quale andrà ad indicare come si classificano quelle celle, ossia il loro grado di frequentazione. Il costruttore andrà quindi a invocare due metodi per il calcolo della classifica, uno per quella della mattina e uno per quella pomeridiana. La logica è la stessa: ricevono in input la mappa "am" (o "pm") dell'accumulatore e trovano le tre celle aventi la lista con dimensione più grande. La dimensione della lista infatti rappresenta il numero di navi che hanno frequentato una relativa cella in una specifica fascia oraria, e le tre dimensioni maggiori si possono quindi associare ai gradi di frequentazione più alti.
Pertanto, nell'output della query2 non sono state riportate soltanto le 3 celle top, ma le **3 liste di celle top**, ossia tutte le celle a pari merito che hanno la frequentazione più alta durante quella finestra temporale. Inoltre sono riportate in

ordine decrescente, quindi la prima lista che si legge è quella al primo posto, e così via.

Il secondo parametro dell'operatore "**aggregate**", ossia **Query2ProcessWindowFunction** ha le stesse funzionalità e la stessa logica di quello nella query1, così come il resto degli operatori che seguono aggregate, ovvero "map" e "addSink". Anche quest'ultimi scrivono il risultato della query sia come stringa da mandare nel sink verso il topic Kafka "QUERY2" sia come file csv.
Dopo la computazione della query2 su una finestra con durata 7 giorni, vengono eseguite nuovamente le stesse operazioni ma su una finestra con durata di 28 giorni.

3.4. Package "query3"

In questo package sono contenute tutte le classi necessarie per la risoluzione della query3. La classe che contiene il flusso logico è "**Query3.java**", che quindi verrà illustrata di seguito.

La classe Query3 ha un solo metodo che si chiama runQuery3, il quale riceve in ingresso il DataStream. Sul flusso di dati viene chiamato l'operatore **keyBy**, che partiziona il flusso di dati in base al tripId. Viene poi applicata una finestra anche qui di tipo tumbling basata su event time con durata di un'ora, ed in seguito viene chiamato l'operatore **aggregate**, che riceve come parametri d'ingresso il costruttore della classe **DistanceAggregate** ed una funzione anonima di tipo ProcessWindowFunction.

- Classe **DistanceAggregate**: questa classe implementa la classe "AggregateFunction" di Flink. Prende in ingresso un oggetto di tipo Ship, come accumulatore un oggetto di tipo **AccumulatorQuery3** e in output un oggetto **OutputDistanceQuery3** (entrambi creati appositamente). La classe **AccumulatorQuery3** ha un attributo chiamato "**positionTimestamp**" di tipo Map<Date, Tuple2<Double, Double», dove chiave Date rappresenta il timestamp dell'oggetto momentaneamente processato, mentre la tupla di <Double, Double> contiene la sua latitudine e longitudine. Inoltre, è presente un metodo chiamato **addNewPosition()** che prende in input il timestamp, la latitudine e la longitudine e li aggiunde alla mappa positionTimestamp. Qui non vengono effettuati controlli per verificare se esiste già un elemento con stessa chiave perché si sta lavorando sempre con lo stesso tripId (grazie alla keyBy) e di conseguenza i timestamps dovrebbero essere tutti diversi fra loro, sia per data che per orario, in quanto i dati riguardano delle navi in movimento. La routine **addNewPosition()** è invocata dal metodo "**add**" della classe **DistanceAggregate**. In quest'ultima, nel metodo "getResult" viene effettuato il calcolo della distanza. Esso riceve in input l'oggetto di tipo AccumulatorQuery3, da cui prende l'attributo positionTimestamp. Le sue chiavi (i timestamps) vengono ordinate (anche se non sarebbe necessario in quanto il dataset è stato ordinato precedentemente), e vengono presi i valori della latitudine e longitudine relativi al primo e all'ultimo timestamp, in modo da poter calcolare la distanza euclidea che separa le due coordinate.

Dopo ciò, avviene l'esecuzione della funzione anonima di tipo **ProcessWindowFunction**, che inserisce il timestamp d'inizio della finestra, chiave e distanza appena calcolata come attributi delle diverse istanze della classe **OutputDistanceQuery3**.
Una volta terminata l'elaborazione dell'operatore aggregate, sono state calcolate le distanze percorse nell'ultima ora per ogni tripId, ma si hanno solo risultati individuali: per accorpare le distanze risultanti in modo da ottenere la classifica dei 5 viaggi con punteggio di percorrenza più alto, viene chiamata una "**windowAll**". In questo modo si può effettuare il merge di cui si ha bisogno. La finestra definita nella window all è anch'essa di tipo tumbling, basata su event time e avente stessa durata della precedente.
Di conseguenza viene chiamato nuovamente l'operatore aggregate, che come parametri prende due funzioni anonime, un'**AggregateFunction** ed una **ProcessAllWindowFunction**.
L'AggregateFunction ha come input **OutputDistanceQuery3**, come accumulatore **FinalAccumulatorQuery3** e come output **FinalOutputQuery3**. Per dare un'idea generale, FinalAccumulatorQuery3 ha come attributo un treeMap Map<Double, String> chiamato "**distanceTripId**" dove la chiave, cioè il double, è dato dalla distanza, mentre il valore è il relativo tripId su cui quella distanza è stata calcolata. Una treeMap è una classe dove le chiavi vengono mantenute in ordine (in questo caso decrescente). Le diverse distanze e relativi tripId di cui si sta facendo il merge vengono quindi aggiunti in questa TreeMap nell'add dell'Aggregatore e poi nel metodo getResult vengono presi i primi 5 valori, in modo da ottenere la top 5 delle distanze percorse, e i relativi trip Id.
Come nelle due query precedenti, seguono una ProcessAllWindowFunction per recuperare la chiave ed il timestamp su cui la finestra ha iniziato a processare, una map e un addSink per inviare i risultati a Kafka e scriverli in formato csv.

4. docker

In questa cartella della repository GitHub sono presenti diversi files tra cui un **Dockerfile** ed un **docker-compose.yml**. Il Dockerfile serve per avviare un container all'interno del quale è presente l'applicazione del producer. Nel dockerfile viene quindi effettuato il setup del container, che prevede un ambiente Ubuntu in cui viene installato java, per poi copiare la jar del file "**connectionToKafka.MyProducer.java**" all'interno del file system del container e poterla eseguire.
Nel docker-compose.yaml sono presenti le immagini dei container da avviare, ossia quello di zookeeper, kafka, e di flink, in particolare del suo job manager (nodo master) e task manager (nodo worker), per il quale è stato specificato uno **scaling** pari a **3**. Infine, nel docker-compose viene effettuata la build del Dockerfile presente nella cartella "docker".
Una volta che il cluster è stato avviato eseguendo lo script "**start-docker.sh**", si può effettuare il submit dei job su flink, ossia dell'applicazione del consumer, come specificato nel file README.md presente nella repository.
La creazione delle jar avviene a valle tramite il comando "**mvn compile assembly:single**".

5. Piattaforme Software

Il sistema è stato realizzato tramite:
- **IntelliJ IDEA Ultimate Edition** come IDE
- Apache **Maven** 3.6.0
- Macchina Virtuale con **Ubuntu 18.04**
- **docker** 20.10.6
- **docker-compose** 1.17.1

6. Risultati dei Tempi d'Esecuzione

I test sono stati svolti utilizzando un processore i7-7700HQ. Le metriche sono state valutate sperimentalmente utilizzando la classe "MetricCalculationSink" presente nel package "utils". "MetricCalculationSink" implementa la classe di flink "SinkFunction" ed è stata utilizzata esclusivamente per il calcolo delle metriche, in sostituzione dell'addSink finale all'interno di runQuery1,2, 3, in cui veniva creato un nuovo FlinkKafkaProducer per l'invio dei dati a Kafka. Il concetto alla base della computazione è che la prima volta che una window finisce di processare i dati che contiene, e quindi entra in **MetricCalculationSink** viene salvato il tempo corrente come "startTime". Dalla seconda volta, viene incrementato un contatore che rappresenta il numero di tuple processate, e calcolato un currentTime dato dalla differenza fra il tempo attuale e lo startTime. Vengono quindi calcolate la **latenza** come rapporto fra currentTime e count e il **throughput** come differenza fra il count e currentTime. Il valore complessivo per il processamento di una query quindi è dato dall'ultimo valore di latenza e throughput che viene stampato. Il tempo è stato misurato in secondi.

Query 1	Throughput Query 1	Latency Qry 1
7 giorni	48,9	0,02
28 giorni	24,5	0,04
ratio 7/28	2,0	0,5

Figure 1. Query1

Query 2	Throughput Query 2	Latency Qry 1
7 giorni	2,3	0,44
28 giorni	0,7	1,34
ratio 7/28	3,0	0,3

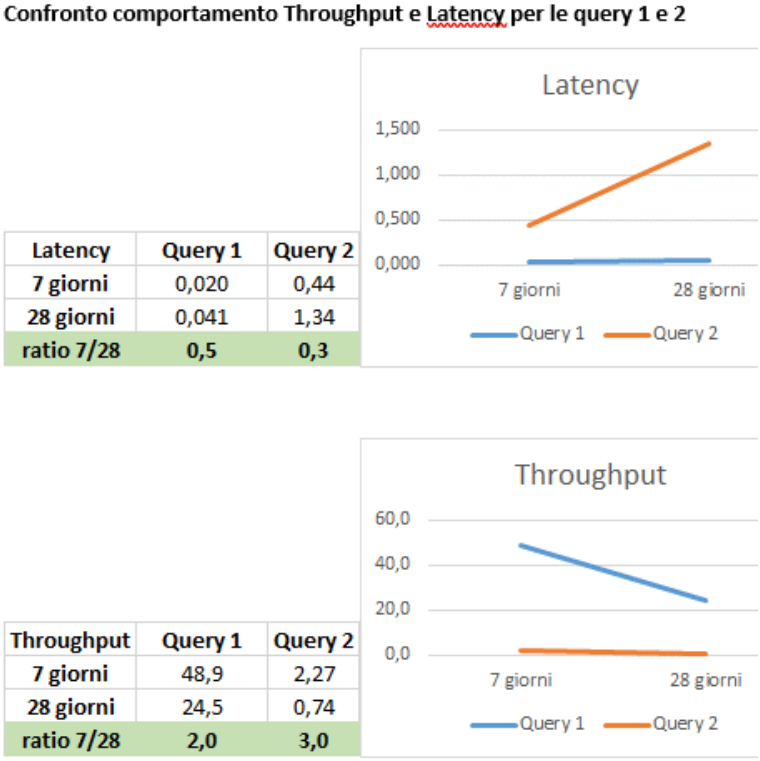
Figure 2. Query2

Query 3	Throughput Query 3	Latency Qry 3
1 ora	87,3	0,01
2 ore	39,8	0,03
ratio 1/2	2,2	0,5

Figure 3. Query3

7. Conclusione

L'unico confronto vero e proprio che si può effettuare è esclusivamente tra le query 1 e 2, in quanto le finestre temporali utilizzate sono le stesse. Si può osservare dalle tabelle e dai grafici riportati in **Figure 1**, **Figure 2**, e in **Figure 4**, che passando dalla query1 alla query2 il throughput diminuisce, mentre la latenza aumenta. Per quanto riguarda un confronto esclusivamente all'interno della stessa query (considerando anche la query3 in **Figure 3**), ma tra finestre temporali differente, si può evincere dai dati riportati che in ogni caso, all'aumentare della dimensione della finestra aumenta anche il throughput, e diminuisce la latenza.



Latency	Query 1	Query 2
7 giorni	0,020	0,44
28 giorni	0,041	1,34
ratio 7/28	0,5	0,3

Throughput	Query 1	Query 2
7 giorni	48,9	2,27
28 giorni	24,5	0,74
ratio 7/28	2,0	3,0

Figure 4. Confronto tra Query1 e Query2 su stesse finestre temporali

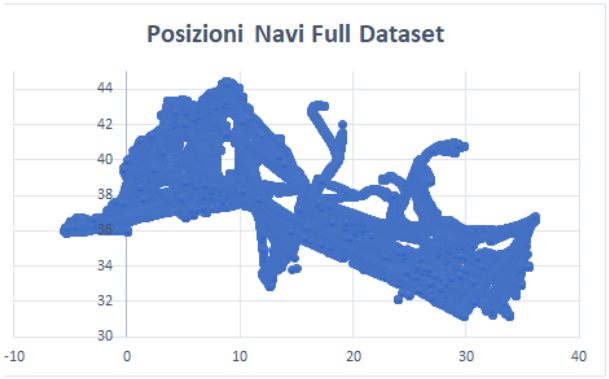


Figure 5. Posizione delle navi senza limitare l'area marittima

References

[1] Apache Flink: "https://ci.apache.org/projects/flink/flink-docs-release-1.13/"

[2] Immagine container kafka: "https://hub.docker.com/r/confluentinc/cp-kafka/"

[3] Immagine container zookeeper: "https://hub.docker.com/_/zookeeper"

[4] Immagine container flink: "https://hub.docker.com/_/flink"