

Architettura per lo Stream Processing in un Cluster

Usando Apache Flink e Apache Kafka

Andrea Paci

Università degli studi di Roma
Tor Vergata
Roma
andrea.paci1998@gmail.com

Alessandro Amici

Università degli studi di Roma
Tor Vergata
Roma
a.amici@outlook.it

ABSTRACT

La seguente trattazione mira a presentare l'architettura sviluppata per il processamento di tipo **Streaming**, in parallelo, all'interno di un cluster.

Verrà innanzitutto presentata l'architettura, descrivendo brevemente ogni singolo componente, per poi entrare nel dettaglio e presentare tutto il sistema, motivando le varie scelte implementative.

Infine, verranno presentati con una breve analisi statistica alcune metriche riguardanti l'esecuzione. A questo seguiranno due brevi paragrafi che trattano i possibili sviluppi futuri e le difficoltà riscontrate durante lo sviluppo.

SCOPO DEL PROGETTO

Il focus principale del progetto è quello di sviluppare un'architettura per effettuare **stream processing** per grandi moli di dati, prodotti in tempi diversi e con una frequenza variabile. In particolare, il sistema ha il compito di rispondere ad alcuni quesiti riguardanti l'*analisi di rotte marittime*, al fine di ricavare dati statistici ed informazioni utili per descrivere la situazione *geospatiale* marittima, nello specifico del *Mar Mediterraneo*. I dati *geo-spatiali* sono da considerarsi in formato **longitudine e latitudine**. Inoltre, l'area presa in esame è divisa in una **griglia** con **40x10 celle**, espressa con la notazione *lettera/numero* (es. C23).

I dati riguardanti le navi in mare sono stati presi da *Automatic Identification System* (AIS), i quali coprono un arco temporale compreso tra il 10 marzo 2015 al 19 maggio 2015. Nonostante la mole di dati, ed in particolare la frequenza con cui vengono prodotti, non giustifichi l'impiego di un ambiente distribuito di stream processing, le scelte progettuali e le considerazioni fatte durante lo sviluppo non tengono conto di questo fattore, ma simulano un contesto dove è necessario avere questo tipo di infrastruttura.

I quesiti a cui è necessario rispondere sono 3:

1. Prendendo in esame solo il Mar Mediterraneo Occidentale, computare per ogni cella il numero medio giornaliero di navi

che la percorrono, facendo distinzioni tra le diverse tipologie di imbarcazioni. Il quesito va risposto sia per una finestra temporale di una settimana che di un mese.

2. Distinguendo tra il Mar Mediterraneo Occidentale e Orientale, fornire la classifica delle 3 celle più frequentate nelle due fasce orarie *00:00-11:59* e *12:00-23:59*. Il quesito va risposto sia per una finestra temporale di una settimana che di un mese
3. Fornire la classifica in *tempo-reale* dei 5 viaggi con percorrenza più alta. La classifica viene stilata per finestre di *un'ora e 2 ore*.

Questi 3 quesiti, insieme a possibili altri, permettono di descrivere il traffico navale non solo nel Mediterraneo, per consentire una gestione dinamica ed automatizzata di rotte e porti di destinazione.

ARCHITETTURA

L'architettura utilizzata per la risposta dei 3 quesiti comprende un insieme di framework e tecnologie orientate ai Big Data, interconnessi opportunamente tra loro.

L'architettura è costituita da 3 componenti principali:

- **Apache Kafka** come **Messaging Layer** per fare **ingestion** dei dati. Come **coordinatore distribuito** dei **broker** del sistema viene utilizzato **Apache ZooKeeper**.

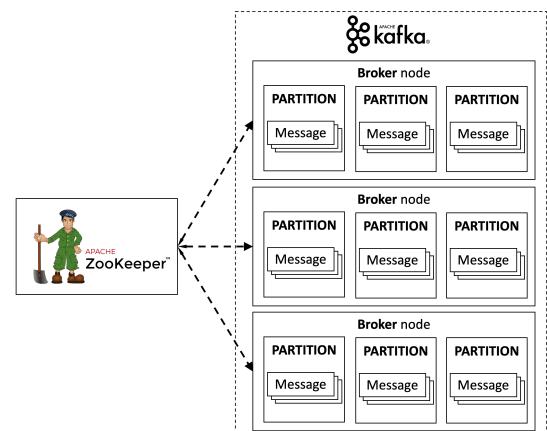


Figura 1: Schematizzazione del cluster Kafka con ZooKeeper

- Apache Flink per il layer di *processing* con Prometheus per il report delle metriche

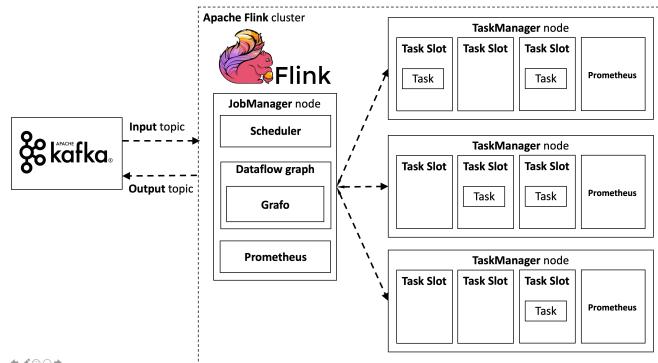


Figura 2: Schematizzazione dell'impiego di Apache Flink

- Applicativo Java che produce i messaggi sui dati navali e legge l'output prodotto da Flink. Inoltre fa il **submit** della **topologia** a Flink

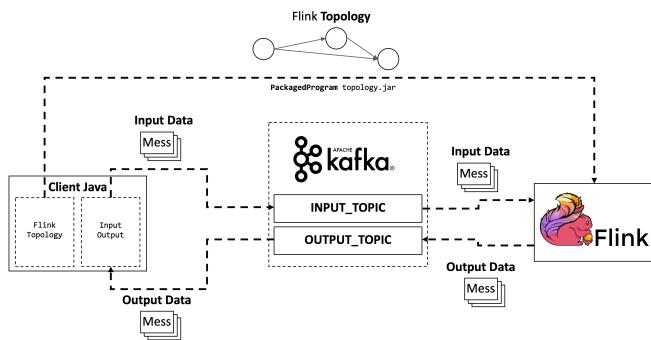


Figura 3: Schematizzazione dell'impiego dell'applicativo Java

1 Apache Kafka

Apache Kafka è una piattaforma open-source per la distribuzione di eventi tramite il pattern **publish-subscribe**. Kafka viene utilizzato sia per l'**ingestion** dei **messaggi** nel sistema, cioè i dati navali, sottoforma di evento con il relativo **timestamp**, sia viene utilizzato come **destinazione** di **output** delle query su Flink. E' quindi responsabile dell'inoltro dei messaggi per il processamento e la raccolta dei dati per l'esportazione in altro formato (es. CSV)

La scelta di usare Kafka risiede nelle seguenti fattori:

- **Supporto nativo** di Flink nell'uso di Kafka come **sorgente**, il quale è in grado di capire il *timestamp* dei messaggi direttamente dai metadati, e come **sink**, entrambe con semantica **exactly-once**.

- **Distribuito e fault tolerant**, che grazie anche alla coordinazione e sincronizzazione ottenuta con ZooKeeper, lo rende adatto per un contesto di esecuzione distribuita **self-managing**.

- **Adatto** per lo **Stream Processing** in quanto è architettato per inviare dati sottoforma di eventi con timestamp.

Il setup di Kafka nel sistema è di *default* con **2 broker** con livello di **partizionamento** pari a 2 e **replicazione** pari a 2. ZooKeeper di *default* è singola istanza. Questi parametri possono essere cambiati nei file di configurazione.

2 Apache Flink

Apache Flink è un engine di **computazione stateful/stateless** per **data stream** basato sull'architettura **master-worker**. Flink è quindi responsabile dell'esecuzione delle 3 query con la conseguente gestione del *parallelismo* e dello *scheduling* tra diversi Task. Il modello di sviluppo di Flink è **funzionale** (es. map, reduce, aggregate,...), il che lo rende adatto allo sviluppo di topologie per **aggregazione** di dati. A Flink viene appoggiato **Prometheus** come sistema di memorizzazione e querying per lo stato e le metriche del cluster di esecuzione, infatti grazie a quest'ultimo è possibile vedere in tempo reale dati come **utilizzazione dei Task Manager** in termini di di CPU e memoria, **latenza** e **throughput** sul singolo operatore.

Di default l'architettura prevede **1 Job Manager** che gestisce le risorse dei nodi di esecuzione, prende in input il *dataflow graph* ed assegna ai Task Manager i vari Task da eseguire. Il cluster di Flink è impostato in **Session Mode**: cioè è in grado di accettare più **Job** (grafi) contemporaneamente.

Oltre al Job Manager, l'architettura prevede **3 Task Manager** con **3 Task Slot** ognuno.

La scelta di **Flink** rispetto ad altri engine come Storm o Heron è da ricercare nelle seguenti caratteristiche:

- **Consistenza** dello stato **exactly-once**
- Gestione e supporto di **event time** e **processing time**
- **Monitoring** built-in
- Gestione di **lateness** grazie ai **watermarks**
- **Checkpointing** asincorno

3 Applicativo Java

L'applicativo Java si occupa di varie funzionalità:

- **Producer:** fa l'ingestion dei messaggi su Kafka, simulando quello che in un sistema di Stream processing fanno le singole sorgenti dei dati (in questo caso, le singole navi). Per questa simulazione i dati sono presi da un dataset preesistente, ma il **delay dell'invio** dei singoli messaggi è **proporzionale all'event time**, in modo da simulare al meglio un contesto reale, ma opportunamente **accelerato**. Grazie al **delay**, i dati vengono inviati con un **ordinamento temporale** come in un caso reale.
- **Consumer:** legge i dati di output delle 3 query su Kafka e genera un file contenente i risultati in formato **csv**. La scrittura su file avviene **progressivamente in append** al fine di gestire casi di **unbounded stream**, dove non è presente la fine del flusso dei dati.
- **Topology submitter:** grazie all'API *PackagedProgram*, invia a Flink la **topologia della rete** ed il **.jar** contenente tutte le dipendenze necessarie per l'esecuzione delle Task. In questo modo è possibile gestire con un singolo eseguibile sia la modifica della topologia e sia il submit, rendendo più semplice e meno oneroso lo sviluppo.

Producer e Consumer scrivono/leggono su due topic separati: un topic di **input** e un topic di **output**, separando così i due flussi.

L'applicativo consente anche un modello di *esecuzione locale*, dove vi è un *Bypass* dell'ingestion layer con Kafka e vi è direttamente l'esecuzione della topologia di Flink con un cluster locale. Questa aggiunta è stata fatta per ridurre ulteriormente i tempi di sviluppo eliminando l'overhead della comunicazione.

Nonostante l'applicativo presenta funzionalità logicamente separate, per una maggiore semplicità sono state inserite in un unico nodo della rete.

Per motivi di testing, è stata introdotta anche una modalità in cui i messaggi vengono inviati **senza delay**: in questo modo è stato possibile provare la **robustezza** della topologia di Flink con una frequenza di dati nel tempo molto alta.

DEPLOYMENT

Il deployment è stato effettuato usando la **containerizzazione dei nodi** del sistema usando **Docker Compose**, il quale automatizza la *build* dei nodi e gestisce le dipendenze *temporali* tra macchine, simulando abbastanza fedelmente un cluster di macchine distinte tra loro. Ogni nodo dell'architettura presentata precedentemente è un container docker all'interno di una Docker Network.

L'architettura è descritta nel **docker-compose.yml** e nei **Dockerfile** presenti nel sorgente. Lo scaling dei nodi, come spiegato precedentemente, ha dei valori di default, ma è possibile modificare questi valori per cambiare le dimensioni del cluster.

Il deployment è completamente automatizzato, infatti tutta l'esecuzione viene iniziata con *docker-compose up*, mentre per interrompere l'esecuzione e pulire l'ambiente è necessario *docker-compose down*. Per rendere l'architettura più portatile possibile sono stati ridotti al minimo il numero di dipendenze con la macchina host, dove l'unica dipendenza rimasta è la presenza di Docker, il resto delle operazioni di configurazione e compilazione vengono processate all'interno dei container. Per motivazioni di sincronizzazione, sono stati aggiunti dei *tempi di attesa* all'interno dell'applicativo per permettere a Kafka ed a Flink il tempo per il setup iniziale.

Il deploy su Docker, a differenza di un deploy sul Cloud con servizi come AWS, permette una maggiore personalizzazione del cluster e un maggior controllo sui singoli nodi. Inoltre, con un deploy manuale è possibile sperimentare la configurazione ed il setup delle varie tecnologie, anche al fine di comprenderne meglio il loro funzionamento.

Nel deploy è presente anche **Grafana** che, collezionando le metriche di Flink esposte da **Prometheus**, permette la visualizzazione di una *dashboard* sulle metriche di Flink *in tempo reale*.

L'utilizzo di Docker su uno stesso nodo rende la latenza di comunicazione tra i vari nodi prossima a 0, ma i singoli nodi, condividendo uno stesso hardware, hanno accesso ad un insieme ristretto di risorse computazionali.

Le tecnologie ed i framework utilizzati sono:

- **Apache Kafka v. 2.8.0**
- **Apache ZooKeeper**
- **Apache Flink v. 1.13.1**
- **Prometheus w/ Oshi-Core e JNA v. 2.27.1**
- **RocksDB** per le metriche di latenza
- **Grafana v. 7.5.7**
- **Java 1.8** (Sia come JDK che come JRE nei nodi del cluster)
- **Docker e Docker-Compose**
- **Conduktor** come client per Kafka

QUERY

L'applicativo sviluppato utilizzando Flink risponde ai 3 quesiti presentati precedentemente.

Prima di eseguire le query, viene caricata la topologia su Flink e successivamente i dati vengono instradati con Kafka. Ogni query ha un **Source** in comune, cioè il **FlinkKafkaConsumer**, responsabile di leggere i dati da Kafka. Dal Source il flusso si *dirama* in **6 flussi: 2 per ogni Query** per diffrenziare le diverse richieste di aggregazione temporale (es. la query 1 va eseguita su di un arco temporale di 7 giorni e 28 giorni). La fine di ogni flusso è segnata dalla presenza di un **Sink**: il **FlinkKafkaProducer**, il

quale invia l'output del flusso al topic di output creato appositamente su Kafka. La topologia della rete così definita permette l'esecuzione delle 3 query in parallelo.

E' presente una fase di **preprocessing** dove viene fatta manipolazione e filtraggio dei dati per prepararli al processamento:

- Conversione al tipo di dato **NavalData**: struttura creata appositamente per racchiudere l'informazione del singolo evento
- **Filtraggio** delle tuple rimuovendo quelle che non rientrano nell'area geografica presa in esame
- **Aaggiunta** dei campi di **cella di appartenenza**, assegnazione al Mar Mediterraneo **Occidentale** o **Orientale** e infine assegnazione allo **slot temporale** prima delle 12:00 o successivo
- **Aaggiunta** del tempo di **lateness** consentito

Per semplicità, l'assegnazione al lato Occidentale o Orientale del Mar Mediterraneo è stata calcolata dividendo l'area di interesse nelle due porzioni tramite una linea verticale posizionata alla sinistra di Trapani. La linea che divide in due aree il Mar Mediterraneo coincide con una "linea" verticale che separa delle celle.

Per aggregare dati su base temporale, ogni query presenta un operatore di **window**: le query 1 e 2 hanno finestre di ampiezza di **7 giorni** e **28 giorni** con **sliding** di un giorno, mentre la terza query ha un'ampiezza di **1 e 2 ore** con sliding factor di 1 ora.

Data la dimensione dell'output generato dall'impiego di una **sliding window** per i 6 flussi in esecuzione, è stato preferito modificare il parametro di sliding tale da utilizzare una **tumbling window** e riportare come output quello generato da questa modifica, anche se ai fini delle richieste sopracitate, è più adeguata una sliding window. Per riportare lo sliding factor a quello originale è sufficiente modificare un parametro di configurazione.

A causa della natura delle finestre in Flink, le quali sono calcolate come *slot fissi* a partire dal 1 gennaio 1970, è stato necessario utilizzare un **offset** per allineare le finestre temporali alla prima tupla del dataset.

Nonostante il conseguente degrado delle prestazioni, ogni query utilizza una **process window function** per formattare lo stream in uscita tale che contenga il *timestamp* di inizio finestra.

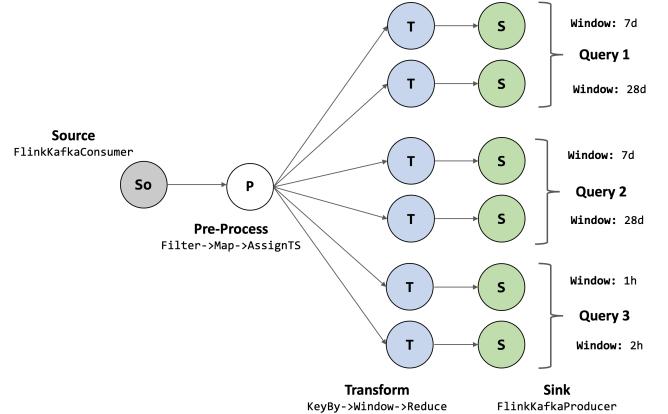


Figura 4: Schematizzazione piano logico di Flink

Di seguito vengono presentati in modo riassuntivo i passaggi delle singole query.

1 Query 1

Come passaggio preliminare viene fatta una **filter** per rimuovere tutte le tuple del Mar Mediterraneo Orientale.

Successivamente, il flusso viene partizionato con **chiave** pari alla **cella di appartenenza** e poi **aggregato**.

La fase di **aggregazione** consiste nel contare le navi **diverse (ship_id)** che giornalmente attraversano la cella su cui si è partizionato il flusso, separando il conteggio a seconda della tipologia di nave.

Infine, un'ultima **map** dove il valore conteggiato precedentemente viene diviso per la grandezza della finestra (7 giorni o 28 giorni).

Timestamp	Cell	SHIP_TYPE=35	Value	SHIP_TYPE=60-69	Value	SHIP_TYPE=70-79	Value	SHIP_TYPE=others	Value
2015/03/10	D1	SHIP_TYPE=35	0.0	SHIP_TYPE=60-69	0.0	SHIP_TYPE=70-79	0.0	SHIP_TYPE=others	0.42857143
2015/03/10	C1	SHIP_TYPE=35	0.0	SHIP_TYPE=60-69	0.0	SHIP_TYPE=70-79	0.0	SHIP_TYPE=others	0.14285715
2015/03/17	D1	SHIP_TYPE=35	0.0	SHIP_TYPE=60-69	0.0	SHIP_TYPE=70-79	0.0	SHIP_TYPE=others	1.0
2015/03/24	D1	SHIP_TYPE=35	0.0	SHIP_TYPE=60-69	0.0	SHIP_TYPE=70-79	0.0	SHIP_TYPE=others	1.0
2015/03/31	D1	SHIP_TYPE=35	0.0	SHIP_TYPE=60-69	0.0	SHIP_TYPE=70-79	0.0	SHIP_TYPE=others	1.1428572
2015/03/31	D2	SHIP_TYPE=35	0.0	SHIP_TYPE=60-69	0.0	SHIP_TYPE=70-79	0.0	SHIP_TYPE=others	0.2857143
2015/04/07	D1	SHIP_TYPE=35	0.0	SHIP_TYPE=60-69	0.0	SHIP_TYPE=70-79	0.0	SHIP_TYPE=others	1.4285715
2015/04/07	D2	SHIP_TYPE=35	0.0	SHIP_TYPE=60-69	0.0	SHIP_TYPE=70-79	0.0	SHIP_TYPE=others	0.71428573
2015/04/07	D3	SHIP_TYPE=35	0.0	SHIP_TYPE=60-69	0.0	SHIP_TYPE=70-79	0.0	SHIP_TYPE=others	0.42857143
2015/04/07	D5	SHIP_TYPE=35	0.0	SHIP_TYPE=60-69	0.0	SHIP_TYPE=70-79	0.0	SHIP_TYPE=others	0.2857143
2015/04/07	H8	SHIP_TYPE=35	0.0	SHIP_TYPE=60-69	0.0	SHIP_TYPE=70-79	0.0	SHIP_TYPE=others	0.14285715

Figura 5: Esempio di output per la prima query con finestra di 7 giorni

2 Query 2

Lo **stream** viene **partizionato** in base alla divisione longitudinale del mare Orientale e Occidentale e con **windowing** di 7 o 28 giorni. **L'aggregazione** accumula i dati navali, considerando le diverse navi soltanto una volta nello stesso giorno, tenendo conto

della **fascia oraria** con due **accumulatori** diversi. Alla fine della finestra viene fatto il conteggio per ogni cella e vengono estrapolate le prime 3 celle con *grado di frequentazione* più alto, differenziando le due fasce giornaliere.

Timestamp	Sea	Time slot	Leaderboard	Time slot	Leaderboard
2015/04/28	WESTERN_MEDITERRANEAN_SEA	BEFORE_NOON	F9;J14;F8;	AFTER_NOON	J14;H8;F9;
2015/04/28	EASTERN_MEDITERRANEAN_SEA	BEFORE_NOON	C20;D20;G33;	AFTER_NOON	C20;D20;G33;
2015/05/05	EASTERN_MEDITERRANEAN_SEA	BEFORE_NOON	C20;D20;G33;	AFTER_NOON	H8;F8;G8;
2015/05/12	WESTERN_MEDITERRANEAN_SEA	BEFORE_NOON	F9;F8;H8;	AFTER_NOON	F8;H8;F7;
2015/05/12	EASTERN_MEDITERRANEAN_SEA	BEFORE_NOON	G33;C20;D20;	AFTER_NOON	G33;C20;D20;

Figura 6: Esempio di output della Query 2 con finestra settimanale.

3 Query 3

La Query 3 si divide in due fasi:

- Nella prima viene calcolato il punteggio di percorrenza tra la prima tupla vista in assoluto e l'ultima tupla all'interno della finestra temporale (di una o due ore) per ogni viaggio (trip_id) diverso. Il punteggio di percorrenza viene calcolato come la distanza *euclidea* tra le coordinate di LAT e LON delle due tuple in esame.
- Nella seconda fase viene estratta la **top 5** dei viaggi che hanno avuto, fino a quel momento, punteggio di percorrenza più alto.

Timestamp	TripID 1	Score	TripID 2	Score	TripID 3	Score	TripID 4	Score	TripID 5	Score
2015/04/11 07:15	0x4e0f1d...09-04-15 13: 8.679045	0x4e0f19...09-04-15 6: 7.962404	0x4e0f5a...10-04-15 12: 3.207172	0x4e0f70...11-04-1 7.8439814E-4	0x4e0f8b...10-04-1 6.607733E-5					
2015/04/11 08:15	0x4e0f1d...09-04-15 13: 8.720733	0x4e0f5a...10-04-15 12: 3.5111074	0x4e0f70...11-04-1 5.002785694E	0x4e0f8b...11-04-1 1.9151016E-5	0x4e0f8b...10-04-1 6.607733E-5					
2015/04/11 11:15	0x4e0f19...09-04-15 6: 8.294712	0x4e0f5a...10-04-15 12: 3.8189024	0x4e0f79...11-04-15 2: 0.0031369505	0x4e0f8b...10-04-1 8.8564186E-5	0x4e0f8b...11-04-1 5.521488E-5					
2015/04/11 13:15	0x4e0f19...09-04-15 6: 8.783308	0x4e0f5a...10-04-15 12: 4.099406	0x4e0f41...11-04-15 1/ 0.302954995	0x4e0f70...11-04-1 0.0239853472	0x4e0f8b...10-04-1 8.628981E-5					
2015/04/11 15:15	0x4e0f19...09-04-15 6: 8.889867	0x4e0f5a...10-04-15 12: 4.392695	0x4e0f40...11-04-15 1/ 0.383970703	0x4e0f70...11-04-1 0.0202013453	0x4e0f8b...10-04-1 8.421607E-5					
2015/04/11 17:15	0x4e0f1d...09-04-15 10: 4.732138	0x4e0f5a...11-04-15 14: 3.8877048	0x4e0f40...11-04-15 15: 0.19821704	0x4e0f70...11-04-1 0.0202013903	0x4e0f8b...10-04-1 8.990123E-5					
2015/04/11 19:15	0x4e0f1d...09-04-15 10: 5.058964	0x4e0f5a...11-04-15 14: 4.889948	0x4e0f40...11-04-15 15: 0.310915912	0x4e0f70...11-04-1 0.0202017907	0x4e0f8b...11-04-1 5.116168E-5					
2015/04/11 21:15	0x4e0f1d...09-04-15 10: 5.379507	0x4e0f5a...11-04-15 14: 2.402398	0x4e0f78...09-04-15 0: 2.0017064	0x4e0f41...11-04-1 0.2891186E	0x4e0f79...11-04-1 0.0038230290					
2015/04/11 23:15	0x4e0f1d...09-04-15 10: 5.707233	0x4e0f5a...11-04-15 14: 2.807973	0x4e0f40...11-04-15 15: 0.49307704	0x4e0f40...11-04-1 0.10579301	0x4e0f79...11-04-1 0.0038811537					
2015/04/12 01:15	0x4e0f1d...09-04-15 10: 6.040908	0x4e0f5a...11-04-15 18: 0.01137095	0x4e0f40...11-04-15 5: 0.38023144	0x4e0f70...11-04-1 0.0208700098	0x4e0f8b...11-04-1 7.04703345E-5					
2015/04/12 03:15	0x4e0f1d...09-04-15 10: 6.435879	0x4e0f5a...11-04-15 5: 0.47319807	0x4e0f79...11-04-15 2: 0.0027806033	0x4e0f40...11-04-1 5.687229E-5	0x4e0f8b...11-04-1 7.04703345E-5					
2015/04/12 05:15	0x4e0f19...09-04-15 10: 6.811007	0x4e0f5a...10-04-15 12: 3.798954	0x4e0f79...11-04-15 2: 0.0029484642	0x4e0f40...11-04-1 5.689195E-5	0x4e0f8b...11-04-1 7.04703345E-5					
2015/04/12 07:15	0x4e0f19...09-04-15 10: 7.197495	0x4e0f5a...10-04-15 12: 7.1048846	0x4e0f41...11-04-15 14: 7.7960897	0x4e0f70...11-04-1 0.0339810184	0x4e0f8b...11-04-1 9.191245E-5					
2015/04/12 09:15	0x4e0f19...09-04-15 10: 7.332581	0x4e0f5a...10-04-15 12: 7.38404	0x4e0f41...11-04-15 14: 8.110301	0x4e0f8b...11-04-1 1.9438314	0x4e0f79...11-04-1 0.0038875028					

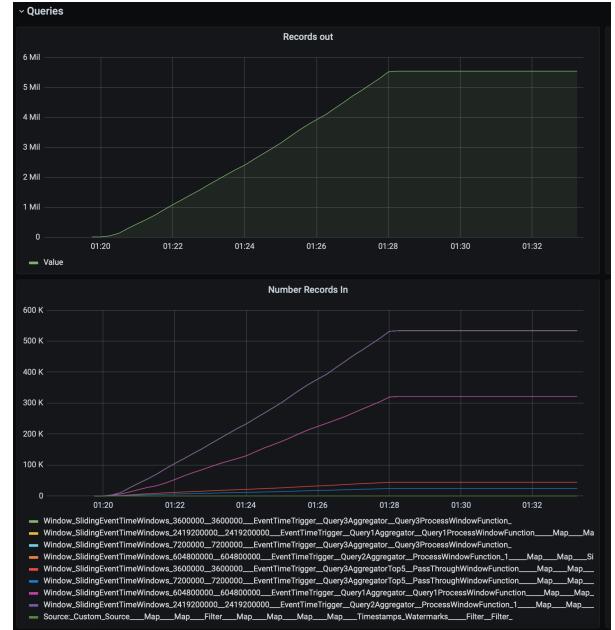
Figure 7: Esempio di output della Query 3 con finestra di due ore.

ANALISI DELLE METRICHE

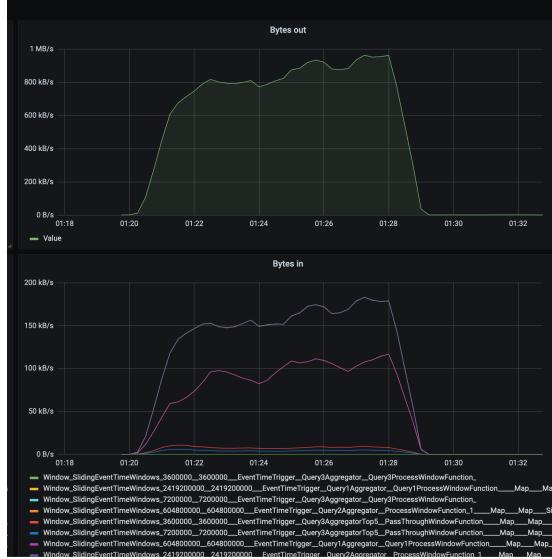
Per dare una valutazione quantitativa sulle performance del cluster, in particolare delle performance di esecuzione con Flink, sono state prese in esame numerose metriche come *task slot in uso*, *memoria/CPU in uso* per *taskmanager/jobmanager*, *Garbage collection overhead* ed infine **throughput** e **latenza**. In un contesto dove vi è **aggregazione dei dati**, specialmente su base temporale, le metriche di throughput e latenza sono fortemente dipendenti da fattori che non rappresentano l'efficienza del sistema. Le metriche, grazie a **Prometheus**, **Oshi-Core**, **JNA** e **Grafana**, sono visualizzabili in una *dashboard* per monitorare in *tempo reale* l'andamento del cluster. I grafici riportati di seguito sono presi dalla Dashboard di Grafana a fine esecuzione.

Il **Throughput** viene presentato sia in **entrata** che in **uscita**, e rappresentato sia come **byte/s** che come **record/s**. Per la distribuzione dei dati in input e per la tipologia delle query, il rapporto tra *throughput* in *entrata* e in *uscita* è estremamente sbilanciato in entrata, infatti da circa 500.000 record in entrata ne escono appena 2.000.

Con questa considerazione, si è pensato che una buona misura di throughput non fosse quella che misura il tasso di uscita dal sistema, bensì il tasso di uscita dai singoli operatori, in modo da avere una visione generale di carico a cui è sottoposto il sistema.



I due grafici mostrano la somma dei record in uscita da ogni operatore e i record in entrata per ogni operatore. Il nodo che ha più record in entrata è il Source insieme ai nodi di preprocessing, mentre il numero di record che in totale attraversano il sistema si attesta sui 5.5 miliardi



Situazione analoga per quanto riguarda il tasso di entrata e uscita, espresso in **byte/s**: Gli operatori che hanno tassi più alti sono quelli di Source e preprocessing, mentre il carico totale a cui sono sottoposti gli operatori arriva fino a 1Mb/s

La **latenza** viene misurata con due modalità diverse:

- Tramite il **LatencyTracker** built-in di Flink e RocksDB
- Tramite **misurazioni Ad-Hoc** prese tramite **Gauge**

Il **LatencyTracker** è uno speciale **record** che viaggia all'interno della topologia e serve a misurare il tempo di attraversamento, senza contare il tempo speso negli operatori, diventando effettivamente una misura di **congestione** nelle code di accesso all'operatore.



Il grafico sulla sinistra riporta l'andamento della latenza sui nodi di **preprocessing**, ma è possibile vedere che il tempo di computazione e attraversamento tra i vari operatori è 0ms a parte per uno spike ad 1ms alla prima injection dei dati.

Il grafico sulla destra invece riporta il **99-percentile** superiore della **latenza** dei singoli nodi, cioè una delle latenze peggiori che sono state registrate. Dal grafico è possibile vedere che mediamente si attesta sui 200ms.

Le **misurazioni di latenza** Ad-Hoc invece misurano mediamente il tempo di processamento degli operatori.



Questi grafici indicano la latenza media delle singole tuple per attraversare gli operatori delle 3 query, per ogni finestra temporale presa in esame. La latenza delle prime due query è più alta a causa della finestra temporale più ampia, e quindi necessariamente una tupla mediamente aspetterà di più prima di uscire dall'operatore. L'andamento crescente dei tempi di latenza è da ricercare nel fatto che la distribuzione dei dati ha una densità maggiore nei mesi di aprile 2015 e maggio 2015.

Proprio come per il throughput, si possono fare osservazioni analoghe per la latenza, infatti il tempo che una tupla impiega per superare un operatore, in questo caso finestrato, dipende dalla grandezza della finestra, il che non misura in alcun modo le performance del sistema. Per porre sotto stress ulteriormente il sistema e vedere come reagisce ad un alto tasso di carico in entrata, tramite un opportuno valore in configurazione, è stato disabilitato il **replay proporzionale** e le tuple sono state inviate senza nessun delay.

Vengono riportate di seguito ulteriori grafici per il monitoring, tra cui Grabage collector e la memoria in uso per TaskManager e JobManager.



SVILUPPI FUTURI

Durante lo sviluppo, sono state pensate alcune soluzioni che potrebbero essere implementate a lungo termine:

- Snellimento della topologia rimuovendo le process window function
- Migliore approssimazione della divisione dei mari
- Tramite ciclo **MAPE**, gestire elasticità del carico

DIFFICOLTA' RISCONTRATE

Le principali difficoltà riscontrate sono state relative al setup del cluster, sia per le documentazioni non particolarmente esplicative in alcuni casi e sia per le forti discrepanze tra una versione ed un'altra dello stesso framework.

Una ulteriore difficoltà è stata misurare le metriche usando i costrutti built-in di Flink, il quale ha bisogno di dipendenze ulteriori per il tracking della latenza.

NOTE FINALI

E' possibile consultare la repository:

<https://github.com/andreapaci/SABD2>

Per consultare il sorgente dell'applicativo ed il setup del cluster.

Nella cartella *Results* sono presenti i risultati delle query in formato *csv*.