Relazione Progetto SABD

Elaborazione dati navali con Apache Storm

Alessio Malavasi
0287437

Gabriele Tummolo
0283629

***Abstract*—Questa relazione descrive il lavoro di elaborazione dei dati riguardo diversi viaggi navali nel mediterraneo effettuati grazie al framework Apache Storm.**

I. Introduzione

Il lavoro realizzato e descritto in questo documento consiste nell' implementazioni di tre query da dati provenienti inizialmente dal file "*prj2_dataset.csv*" caricato sulla repository github del progetto (link github dati).

I dati presenti nel csv non vengono letti tutti insieme ed elaborati, ma viene fatto un replay di questi ultimi in finestre che comprendono dati suddivisi in intervalli di 30 minuti, basate non sul tempo reale di esecuzione, ma sull'event time ricavato da una colonna del file denominata "TIMESTAMP". Il framework principale utilizzato per l'elaborazione dei dati è Apache Storm.

II. Architettura

L'architettura utilizzata per la realizzazione delle query è mostrata il Fig.1 ed è composta da diversi framework. Come prima attività svolta Apache Nifi effettua parte del preprocessing dei dati eliminando alcune colonne non necessarie nell'elaborazione dal file csv "*prj2_dataset.csv*" e l'inserimento all' interno dell'HDFS. Quest'ultimo è utilizzato sia per memorizzare i dati di input, sia in seguito per i risultati delle query espressi sempre in formato csv. A questo punto viene effettuato il replay dei dati e il framework Storm procede all'elaborazione dei dati ed al salvataggio dei risultati all'interno di una coda di RabbitMQ specifica per ogni query. Un lettore esterno infine legge i messaggi dalla coda specifica della query in questione su RabbitMQ e li scrive uno per volta su un file csv all'interno dell'HDFS.

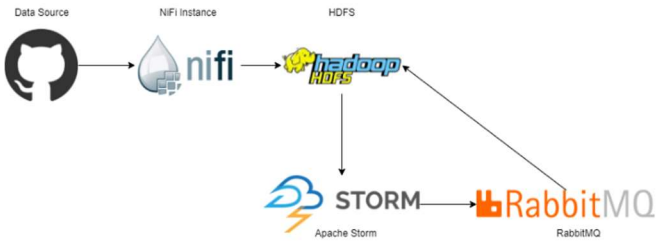

Fig.1 Architettura del Sistema

III. Deploy Sistema

Passiamo ora a descrivere come viene effettuato il deploy dell'architettura utilizzata. L'esecuzione delle query ed i test sono stati eseguiti in locale su una macchina virtuale nell'ambiente di virtualizzazione VirtualBox con 4 cpu-core e 5GB di memoria assegnati. Per effettuare il deploy delle varie parti del sistema basta eseguire il seguente comando una volta posizionatosi nella cartella dist/ all'interno della cartella del progetto:

```
1.  sh ./startEnvironment.sh
```

Una volta eseguito, ci si può collegare al container contenente la CLI di Storm ed eseguire una delle tre query presenti nelle diverse cartelle:

- /data/query1
- /data/query2
- /data/query3

I comandi per eseguire le query sono i seguenti:

```
1.  sudo docker attach storm-cli
2.  cd /data/query1(2/3)
3.  sh ./launchTopologyLocal.sh
```

I file che servono a istanziare la topologia inoltre permettono di inserire un parametro che determina la finestra temporale dell'elaborazione dei dati. Per la prima e la seconda query i parametri inseribili sono "week" e "month", ed indicano rispettivamente una finestra di una settimana o di un mese. Per la terza query invece i parametri inseribili sono "1" o "2" che indicano una finestra di una o due ore.
Infine per terminare l'esecuzione e eseguire lo shutdown di tutti i componenti, si può eseguire il seguente script all'interno della cartella /dist del progetto:

```
1.  sh ./stopEnvironment.sh
```

IV. Apache NiFi

Come già esposto in precedenza, il framework utilizzato per recuperare i dati automaticamente dalla sorgente ed inserirli all' interno dell'HDFS è Apache NiFi la cui configurazione è visibile in Fig.2. Inoltre NiFi effettua anche il preprocessamento dei dati eliminando alcune colonne non necessarie per l'elaborazione dei dati, riconducendo ogni riga del csv al seguente formato:

SHIP_ID, SHIPTYPE, LON ,LAT, TIMESTAMP, TRIP_ID

I processori presenti all'interno dello schema NiFi sono i seguenti:

- *GetHTTP*: processore necessario per scaricare il file csv completo da una sorgente http
- *ReplaceText*: processore che permette di sostituire o eliminare delle colonne all'interno di un file csv. Nel nostro caso è stato utilizzato per eliminare le colonne non necessarie
- *PutHDFS*: processore che permette di inserire file all'interno di una specifica cartella dell'HDFS, in questo caso /input.

Dopo aver effettuato il deploy di tutti i framework, la web UI di NiFi sarà visibile, dopo qualche minuto, all' indirizzo

http://localhost:9999/nifi/ ed il NiFi dataflow sarà eseguito automaticamente.

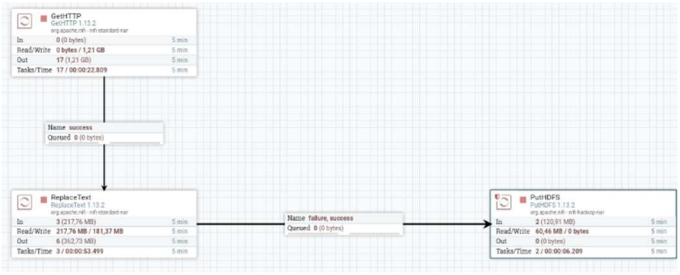

Fig.2 Nifi dataflow

V. HDFS

Nella realizzazione del Progetto è stato utilizzato HDFS come file system distribuito per il salvataggio dei dati di input e di output delle diverse query. Il file system è organizzato con la seguente suddivisione in directory:

- /input: dove vengono salvati i file di input originari e quelli che hanno subito il preprocessing
- /results: dove sono contenuti i risultati delle query in formato csv

Una volta eseguito il deploy dei componenti tramite l'apposito script, lanciata la topologia di una delle query disponibili con il suo relativo lettore di RabbitMQ, è possibile osservare i risultati ottenuti utilizzando il seguente commando:

```
1. hdfs dfs -cat /results/<nome-file-output>.csv
```

I nomi dei file di output prodotti sono i seguenti:

- Query1Week.csv
- Query1Month.csv
- Query2Week.csv
- Query2Mont.csv
- Query3One.csv
- Query3Two.csv

Il nome dei file di output comunque è configurabile all'intero dell'apposito script che esegue il consumatore esterno di RabbitMQ.

VI. RABBITMQ

Come accennato in precedenza, al termine dell' esecuzione di una query, i risultati vengono inseriti sottoforma di messaggi all' interno di apposite code diverse per ciascuna query:

- query1
- query2
- query3

Per osservare i risultati prodotti ed inseriti come messaggi all'interno della coda di RabbitMQ, da locale si può eseguire il seguente script (presente nella corrispettiva cartella della query di cui si vuole osservare il risultato):

```
1. sh ./launchMonitor <nome_coda>
   <nome_file_output>
```

Una volta lanciato il lettore questo andrà a mostrare sul terminale ognuno dei messaggi letti, che corrisponde a una riga del csv dei risultati, ed a scriverlo inoltre all'interno di un file posizionato nella cartella /results dell'HDFS. Questo passaggio attraverso RabbitMQ è stato realizzato in ciascuna query attraverso degli appositi Bolt replicati chiamati "RabbitMQExporterBolt".

E' stato necessario effettuare questo passaggio in quanto le righe contenenti i risultati delle query arrivano contemporaneamente da varie repliche dei Bolt precedenti ed avrebbe rallentato troppo il sistema avere un bolt singolo che si occupasse direttamente della scrittura su file.

VII. ELABORAZIONE DATI

Passiamo ora alla realizzazione vera e propria delle query assegnate.

A. Preprocessing

Come primo procedimento è stata effettuata una semplice fase di preprocessing dei dati ordinando il csv "*prj2_dataset.csv*" rispetto alla colonna del timestamp in ordine temporale crescente. Per fare ciò è stato realizzato un semplice script in java che legge il file csv e inserisce ogni riga all'interno di una lista convertendo il timestamp dal formato "*dd-MM-yy HH:mm*" al formato "*yy-MM-dd HH:mm*". In questo modo si è ordinato la lista su questo parametro in maniera lessicografica ed infine si è riscritta la lista così ordinata in un file chiamato "*dataset_sorted.csv*". Lo stesso processo è stato realizzato e provato anche utilizzando il framework Storm, ma per ottenere lo stesso risultato l'elaborazione è risultata notevolmente più lenta. Probabilmente, per la dimensione comunque limitata del file da processare, l' overhead aggiunto per l'istanziazione della topologia di Storm non rende conveniente l'esecuzione all'interno di quest'ultimo.

B. Query 1

Per la realizzazione della prima query, utilizzando il file "*data_sorted.csv*", sono state utilizzate solo le seguenti colonne:

- SHIPID: identificativo della nave a cui si riferisce la riga del csv
- SHIPTYPE: numero che identifica la tipologia della nave
- LON: longitudine in cui si trova in quell'istante la nave
- LAT: latitudine in cui si trova in quell'istante la nave
- TIMESTAMP: istante di tempo reale in cui è stato registrato un determinato dato.

A partire dalla Fig.3, andiamo ora a descrivere la composizione della topologia istanziata con Storm e le funzionalità di ciascun suo componente.

- *ReaderCSVSpout:* componente che legge i dati dal file csv nell'HDFS e li invia al componente successivo. Leggendo il dataset ordinato, i dati che invia sono temporalmente ordinati

- *SectorCoverterBolt:* grazie all'utilizzo di una TumblingWindow a questo componente arrivano dati raggruppati in finestre temporali di 30 minuti basati sull' event time presente nei dati. Una volta ricevuta una finestra si occupa di convertire le coordinate in una stringa che indica il settore di dove si trova la nave in questione. I settori sono suddivisi in Mar Mediterraneo occidentale ed orientale con la seguente suddivisione:
    - *Range lon occidentale :* [-6.0 ; 12.27]
    - *Range lon orientale :* (12.27 ; 37.0]
    - *Range lat occidentale e orientale: [32.0 ; 45.0]*

  Se il settore appartiene al Mar Mediterrano orientale non inoltra la tupla.
- *CountBolt:* bolt con stato. Funziona da aggregatore giornaliero dei dati e si occupa di effettuare il conteggio delle navi diverse che sono presenti in un settore ogni giorno, e poi emette una tupla con il conteggio all'interno.
- *SumBolt:* bolt replicati con stato. Ogni replica si occupa di alcuni settori in quanto le tuple vengono raggruppate in base al campo "*sector_id*" ed accumula i dati giornalieri di ogni settimana o mese (a seconda della configurazione). Una volta arrivati tutti i dati, somma ed effettua la media ed infine emette la tupla come una stringa unica formattata in maniera tale da poter essere già scritta in un file csv.
- *RabbitMQExporterBolt:* bolt replicati ai quali arrivano delle tuple suddivise casualmente e si occupano semplicemente di inoltrare le stringhe al loro interno in una coda specifica di RabbitMQ chiamata "*query1*".

Una volta fatta partire la topologia dallo storm-cli container, da locale il lettore esterno legge i dati dalla coda RabbitMQ e li scrive direttamente su un file csv all'interno dell HDFS, oltre che a stamparli sul terminale.
Ogni riga del csv riporta i risultati nel seguente formato:

timestamp, sector_id, militari, passeggeri, cargo,other

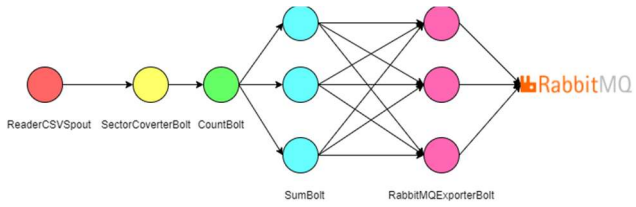

Fig.3 Topologia Query 1

C. Query2

Per la realizzazione della seconda query, utilizzando il file "*data_sorted.csv*", sono state utilizzate solo le seguenti colonne:

- SHIPID: identificativo della nave a cui si riferisce la riga del csv
- LON: longitudine in cui si trova in quell'istante la nave
- LAT: latitudine in cui si trova in quell'istante la nave
- TIMESTAMP: istante di tempo reale in cui è stato registrato un determinato dato.

A partire dalla Fig.4, andiamo ora a descrivere la composizione della topologia istanziata con Storm e le funzionalità di ciascun suo componente.

- *ReaderCSVSpout:* componente che legge i dati dal file csv nell' HDFS e li invia al componente successivo. Leggendo il dataset ordinato, i dati che invia sono temporalmente ordinati
- *SectorCoverterBolt:* grazie all'utilizzo di una TumblingWindow a questo componente arrivano dati raggruppati in finestre temporali di 30 minuti basati sull' event time presente nei dati. Una volta ricevuta una finestra si occupa di convertire le coordinate in una stringa che indica il settore di dove si trova la nave in questione. Inoltre, come campo aggiuntivo della tupla emessa, inserisce anche un valore che indica se tale tupla si riferisce a Mar Mediterraneo occidentale o orientale.
- *CountBolt:* bolt con stato. Funziona da aggregatore giornaliero dei dati e si occupa di effettuare il conteggio delle navi diverse che sono presenti in un settore ogni giorno, e poi emette una tupla con il conteggio all'interno. L' emissione dei dati viene fatta al termine di ogni giorno (sempre basato sull'event time) ed è suddivisa in due fasce che vanno da [00:00 - 11:59] e [12:00 – 23:59]
- *SumBolt:* bolt replicati con stato. Ogni replica si occupa di alcuni settori in quanto le tuple vengono raggruppate in base al campo "*sector_id*" ed accumula i dati giornalieri di ogni settimana o mese (a seconda della configurazione). Una volta arrivati tutti i dati, somma le presenze giornaliere in ogni settore ed a fine tempo di attesa (settimanale o mensile) emette la tupla.
- *RankBolt:* due bolt replicati con stato. Ad ognuno vengono affidati i dati settimanali o mensili di un settore di mare (occidentale o orientale), e si occupano di ordinare tutti i settori in base al numero di navi diverse presenti. Una volta ordinate restituisce i primi tre settori con più navi presenti per ogni zona di mare e per ogni fascia oraria.
- *RabbitMQExporterBolt:* bolt replicati ai quali arrivano delle tuple suddivise casualmente e si occupano semplicemente di inoltrare le stringhe al loro interno in una coda specifica di RabbitMQ chiamata "*query2*".

Una volta fatta partire la topologia dallo storm-cli container, da locale il lettore esterno legge i dati dalla coda RabbitMQ e li scrive direttamente su un file csv all'interno dell HDFS, oltre che a stamparli sul terminale.
Ogni riga del csv riporta i risultati nel seguente formato:

timestamp, mare, prima, sect#1-sect#2-sect#3,

seconda, #sect1-#sect2-#sect3

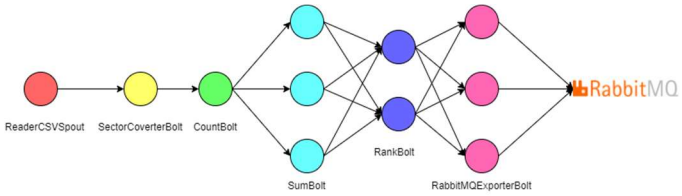

Fig.4 Topologia Query 2

D. Query3

Per la realizzazione della seconda query, utilizzando il file "*data_sorted.csv*", sono state utilizzate solo le seguenti colonne:

- TRIP_ID: identificativo del viaggio comprendente parte di SHIPID e data di partenza e data di presunto arrivo
- LON: longitudine in cui si trova in quell'istante la nave
- LAT: latitudine in cui si trova in quell'istante la nave
- TIMESTAMP: istante di tempo reale in cui è stato registrato un determinato dato.

A partire dalla Fig.5, andiamo ora a descrivere la composizione della topologia istanziata con Storm e le funzionalità di ciascun suo componente.

- *ReaderCSVSpout:* componente che legge i dati dal file csv nell' HDFS e li invia al componente successivo. Leggendo il dataset ordinato, i dati che invia sono temporalmente ordinati
- *DistanceBolt:* grazie all'utilizzo di una TumblingWindow a questo componente arrivano dati raggruppati in finestre temporali di 30 minuti basati sull' event time presente nei dati. Una volta ricevuta una finestra questo componente si occupa di calcolare la distanza euclidea percorsa in un determinato viaggio tra le coordinate attuali e le ultime coordinate note dello stesso viaggio. Alla fine di ogni intervallo di tempo (un ora o due ore configurabile all'avvio della topologia) emette quindi una serie di tuple che, per ogni viaggio, contengono la somma della distanza percorsa. Se un viaggio termina all'interno dell'ora considerata viene considerato valido all'interno di quest'ultima ma non considerato nella successiva.
- *PartialRankBolt:* bolt replicati. Ad ognuno vengono affidati i dati emessi dal DistanceBolt raggruppati per TRIP_ID. Ognuno di essi quindi si occupa di ordinare solamente alcuni dei viaggi per una determinata fascia oraria ed in seguito inoltra tali dati ordinati. Questo bolt ha lo scopo di alleggerire il lavoro del bolt successivo che non può essere replicato.
- *GlobalRankBolt:* bolt unico con stato. A questo bolt arrivano i dati parzialmente ordinati delle varie fasce orarie. Il suo scopo è quello di completare l'ordinamento ed emettere una tupla contenente i primi cinque viaggi per distanza percorsa.
- *RabbitMQExporterBolt:* bolt replicati ai quali arrivano delle tuple suddivise casualmente e si occupano semplicemente di inoltrare le stringhe al loro interno in una coda specifica di RabbitMQ chiamata "*query3*".

Una volta fatta partire la topologia dallo storm-cli container, da locale il lettore esterno legge i dati dalla coda RabbitMQ e li scrive direttamente su un file csv all'interno dell HDFS, oltre che a stamparli a schermo.
Ogni riga del csv riporta i risultati nel seguente formato:

timestamp, trip1, dist1, trip2, dist2, trip3, dist3,
trip4, dist4, trip5, dist5

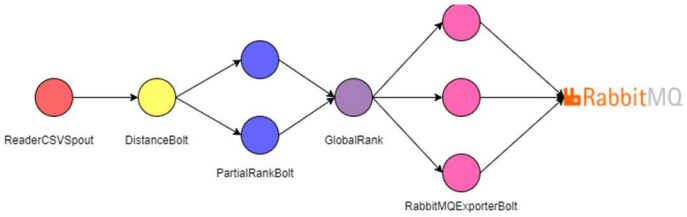

Fig.5 Topologia Query 3

VIII. TEST E PRESTAZIONI

Per quanto riguarda i test e le prestazioni delle diverse query sono state prese in considerazione due metriche principali:

- *Throughput:* il numero medio di tuple al secondo emesse dal sistema (quindi le tuple emesse dai RabbitMQExporterBolt).
- *Laterncy:* il tempo di latenza delle tuple in ciascun bolt. Per latenza si intende il tempo trascorso tra l'ingresso di una tupla nel nodo e l'uscita di un'altra.

Per quanto riguarda la misurazione della prima metrica non ne abbiamo trovata una adatta esattamente ai nostri scopi, per questo ne abbiamo creata una personalizzata chiamata "*Throughput*" e la abbiamo registrata nella topologia grazie alle apposite API fornite da Storm (registerMetric()). Quello che fa questa nuova metrica non è altro che restituire ogni 30 secondi il numero di volte che è stato eseguito il metodo "execute" del bolt alla quale viene applicata, nel nostro caso RabbitMQExporterBolt per tutte e tre le query.
Per quanto riguarda la seconda metrica anche questa è stata creata personalizzata in quanto la metrica "execute-latency" già implementata in Storm ha una grana in secondi, poco precisa per poter contare il tempo trascorso tra una tupla entrata ed una tupla emessa in un nodo, ed infatti restituisce molto spesso un valore pari a zero secondi. La metrica personalizzata invece è stata realizzata utilizzando il timer java in nanosecondi.
Per avere accesso a questi dati, è stata creata per ogni query una classe chiamata "MetricConsumer" che estende l'interfaccia fornita da Storm "IMetricsConsumer". Questa classe è istanziata come se fosse un altro bolt della topologia al quale vengono mandate periodicamente queste metriche

sottoforma di Collection<DataPoint> e si possono gestire come meglio si crede. Nel nostro caso i dati così ricavati vengono salvati su un file csv, diverso per ogni query, per poi essere in seguito elaborati ed analizzati.
L'analisi dei dati del Throughput viene effettuata sommando i valori presenti nel file di output ogni tre (numero di RabbitMQExporter che mandano i dati singolarmente) ed in seguito il risultato di ogni somma diviso per 30 (numero di secondi nell'intervallo di osservazione) per ottenere il numero di tuple emesse al secondo. Infine viene calcolato l'intervallo di confidenza della media al 95% per ottenere una stima della media del throughput del sistema.
L'analisi dei dati di Latency invece avviene semplicemente calcolando l'intervallo di confidenza per la media dei tempi presenti sul file di output per ogni configurazione possibile di replicazione dei bolt provata e per ciascun bolt.
I risultati cosi ottenuti vengono infine riportati su grafici per vedere l'andamento di tali valori al variare delle replicazioni provate.

A. Test query1

Per quanto riguarda la query1 i bolt replicati sono i SumBolt ed è stata misurata la latenza dei bolt utilizzando un grado replicazione di quest'ultimo che varia da uno a tre.
Di seguito, nelle tabelle e in Fig.6, mostriamo i risultati ottenuti per ciascuna metrica.
I risultati per la metrica della latenza sono espressi in millisecondi (ms) mentre per il throughput in tuple emesse al secondo dal sistema.

Osservando i risultati ottenuti possiamo concludere che il grado di replicazione che minimizza la latenza per entrambe le esecuzioni è quello con il SumBolt replicato tre volte.
Questo risultato è in linea con quanto atteso in quanto aumentando la replicazione di questo nodo il carico di lavoro viene diviso tra le diverse repliche.
Il throghput minore dell'esecuzione mensile è dovuto a un minor numero di tuple da emettere come output rispetto a quella settimanale.

week	**Sum 1**	**Sum 2**	**Sum 3**
Sector	5.95 (**)	**	**
Count	3212.57 (--)	--	--
Sum	0.0204	0.01981	0.01158
Throughput	6.47	6.47	5.66

Tabella 1 Latenza media bolt per esecuzione settimanale

month	**Sum 1**	**Sum 2**	**Sum 3**
Sector	**	**	**
Count	--	--	--
Sum	0.04714	0.05485	0.04413
Throughput	1.13(++)	++	++

Tabella 2 Latenza media bolt per esecuzione mensile

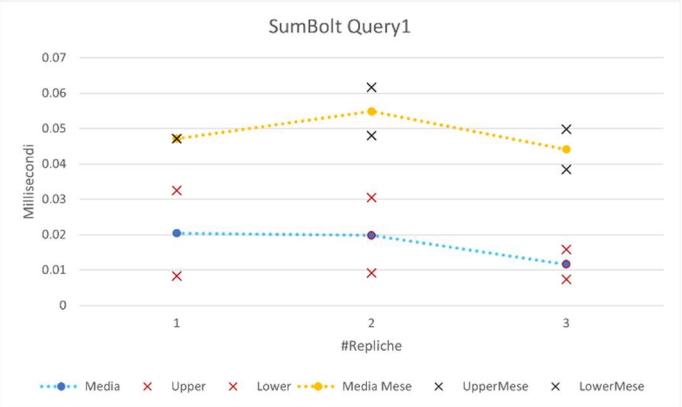

Fig.6 Risultati replicazione SumBolt

B. Test query2

Per quanto riguarda la query2 i bolt replicati sono i SumBolt e i RankBolt. Per i SumBolt si è provato a prendere la latenza dei bolt utilizzando un grado replicazione di quest'ultimo che varia da uno a tre.
Per quanto riguarda RankBolt invece il grado di replicazione è stato scelto pari a due in quanto ognuno dei due bolt si occupa di uno delle due zone di Mar Mediterraneo.
I risultati ottenuti sono mostrati nelle tabelle e in Fig.7.
Per quanto riguarda il grado di replicazione del SumBolt abbiamo scelto di porlo pari a due. In entrambe le modalità di esecuzione, infatti, è vero che la latenza scende al passaggio da uno a due bolt per poi rimanere costante, ma lo stesso non si può dire per i due bolt Rank. Questi ultimi infatti presentano il tempo di latenza con il suo valore ottimale con un grado di replicazione del SumBolt pari a due. Probabilmente un grado di replicazione troppo alto porta in sovraccarico di lavoro uno dei due RankBolt (o entrambi) soprattutto nella modalità di esecuzione settimanale dove in generale le tuple emesse sono infatti più numerose.

week	**Sum 1**	**Sum 2**	**Sum 3**
Sector	9.951 (--)	--	--
Count	2651.24(**)	**	**
Sum	0.00152	0.00127	0.00137
Rank2	0.2388	0.1914	0.6708
Throughput	0.08	0.06	0.07

Tabella 3 Latenza media bolt per esecuzione settimanale

month	**Sum 1**	**Sum 2**	**Sum 3**
Sector	--	--	--
Count	**	**	**
Sum	0.0031	0.00223	0.00217
Rank2	0.8817	0.8987	0.89915

Tabella 4 Latenza media bolt per esecuzione mensile

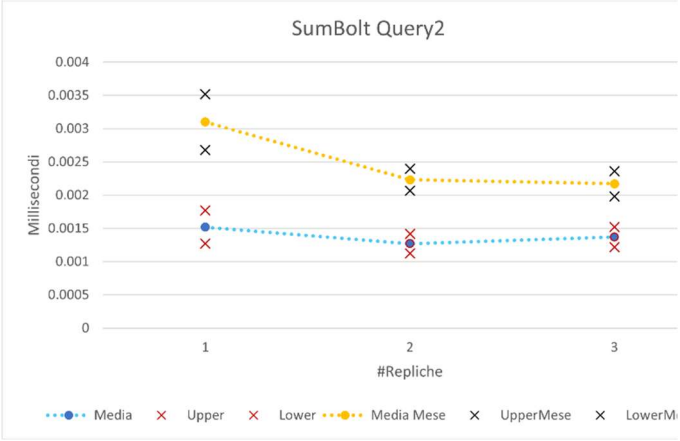

Fig.7 Risultati replicazione SumBolt

C. *Test query3*

Per quanto riguarda la query3 l'unico bolt replicato è PartialRankBolt che si occupa dell'ordinamento parziale dei settori. Il GlobalRankBolt non può essere replicato altrimenti non potrebbe fare l'ordinamento totale.
I risultati ottenuti sono visibili nelle due tabelle e in Fig.8
Per quanto riguarda il grado di replicazione del PartialBolt abbiamo scelto di porlo pari a due. Se è vero infatti che nella modalità di esecuzione oraria la latenza di questo bolt diminuisce con il passaggio da due a tre repliche, nell'altra modalità invece aumenta. Inoltre in entrambi i casi invece la latenza del GlobalBolt aumenta probabilmente a causa del fatto che ha più parti di lista da ordinare tutte con pochi elementi, cosa che peggiora le prestazioni ed appesantisce il sistema invece di migliorarlo.
Per quanto riguarda il throughput come in precedenza quello della modalità bi-oraria è minore ma solamente per il fatto che ci sono meno tuple da emettere rispetto a quella oraria.

1 h	**PartialBolt2**	**PartialBolt3**
DistanceBolt	37.02(--)	--
PartialBolt	71.25	57.18
GlobalBolt	0.119	0.798
Throughput	7.17(**)	**

Tabella 5 Latenza media bolt per esecuzione oraria

2 h	**PartialBolt2**	**PartialBolt3**
DistanceBolt	103.29(++)	++
PartialBolt	165.80	292.03
GlobalBolt	0.346	0.754
Throughput	3.49(**)	**

Tabella 6 Latenza media bolt per esecuzione bi-oraria

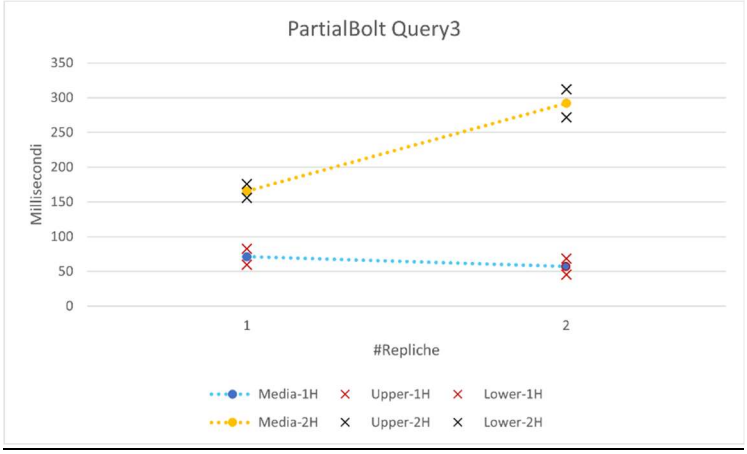

Fig.8 Risultati replicazione PartialBolt

IX. RIFERIMENTI

A. *Link utili*

Riportiamo in seguito i riferimenti utilizzati per la realizzazione del Progetto:

- NiFi doc https://nifi.apache.org/docs.html
- Hadoop doc https://hadoop.apache.org/docs/stable
- Storm doc http://storm.apache.org/releases/2.2.0/index.html
- RabbitMQ doc https://www.rabbitmq.com/documentation.html
- Docker compose https://docs.docker.com/