Sistemi e Architetture per Big Data - AA 2019/2020

Progetto 2: Analisi dei ritardi e guasti del trasporto scolastico di NYC con Flink

Marco Balletti
Dipartimento di Ingegneria dell'Informazione
Università degli studi di Roma "Tor Vergata"
Roma, Italia
marco.balletti@alumni.uniroma2.eu

Francesco Marino
Dipartimento di Ingegneria dell'Informazione
Università degli studi di Roma "Tor Vergata"
Roma, Italia
francesco.marino.412@alumni.uniroma2.eu

ABSTRACT

Questo documento riporta i dettagli implementativi riguardanti l'analisi mediante Flink e Kafka Streams del *dataset* contenente informazioni su ritardi e guasti del trasporto scolastico nella città di New York dal 1 settembre 2015 al 27 novembre 2019. Verrà descritta, insieme a tali dettagli implementativi, anche l'architettura a supporto dell'analisi.

KEYWORDS

Scuola, Bus, Ritardi, Compagnie, New York City, Statistiche, Data Stream Processing, Flink, Kafka, Kafka Streams, Docker

1 Introduzione

L'analisi effettuata si pone lo scopo di valutare delle statistiche relative a ritardi e guasti degli autobus che forniscono il servizio di trasporto scolastico per New York City.

Il *dataset* preso in considerazione è `bus-breakdown-and-delays` (disponibile all'URL [1]) che contiene i dati riguardanti i disservizi riscontrati nel trasporto scolastico a NYC dal 1 settembre 2015 al 27 novembre 2019. Il *dataset*, campionato con la granularità del minuto, si compone dei seguenti campi: `School Year`, `Busbreakdown ID`, `Run Type`, `Bus No`, `Route Number`, `Reason` (motivazione del disservizio), `Schools Serviced`, `Occurred On` (nel formato yyyy-MM-ddThh:mm:ss.SSS), `Created On` (nel formato yyyy-MM-ddThh:mm:ss.SSS), `Boro` (stringa di caratteri rappresentante il quartiere o la contea), `Bus Company Name`, `How Long Delayed` (stringa di caratteri rappresentante il ritardo), `Number Of Students On The Bus`, `Has Contractor Notified Schools`, `Has Contractor Notified Parents`, `Have You Alerted OPT`, `Informed On`, `Incident Number`, `Last Updated On`, `Breakdown or Running Late` e `School Age or PreK`.

L'analisi, in particolare, si compone delle tre *query* elencate di seguito.

1.1 Query 1

Per la prima *query* si richiede di calcolare il ritardo medio degli autobus per quartiere su finestre temporali giornaliere, settimanali e mensili.

L'output della *query* deve essere riportato secondo il formato `ts, boro_x, avg_x, … , boro_z, avg_z` dove `ts` rappresenta il *timestamp* relativo all'inizio del periodo su cui è calcolata la media

1.2 Query 2

Per la seconda *query* si richiede di fornire la classifica delle tre cause di disservizio più frequenti (ad esempio *Heavy Traffic*, *Mechanical Problem*, *Flat Tire*) nelle due fasce orarie di servizio 5:00-11:59 (`slot_a`) e 12:00-19:00 (`slot_p`) su finestre temporali giornaliere e settimanali.

L'output della *query* deve essere riportato secondo il formato `ts, slot_a, rank_a, slot_p, rank_p` dove `ts` rappresenta il *timestamp* di inizio classifica.

1.3 Query 3

L'ultima *query* richiede di fornire la classifica, su finestre temporali giornaliere e settimanali, delle 5 compagnie che hanno il punteggio di disservizio più alto. Tale punteggio viene calcolato come la somma pesata del numero di ritardi dovuti a *Heavy Traffic*, *Mechanical Problem* e *Other Reason* (tutte le cause diverse da *Heavy Traffic* e *Mechanical Problem* sono classificate come *Other Reason*) in base alla formula $w_t t + w_m m + w_o o$ dove t, m, o rappresentano il numero di ritardi dovuti rispettivamente a *Heavy Traffic*, *Mechanical Problem* e *Other Reason*. I valori dei parametri considerati sono: $w_t = 0.3$, $w_m = 0.5$ e $w_o = 0.2$; si tiene, inoltre, conto che, se la durata del ritardo eccede i 30 minuti, questo deve essere conteggiato due volte.

L'output della *query* deve essere riportato secondo il formato `ts, vendor_1, rating_1, vendor_2,`

`rating_2`, … , `vendor_5`, `rating_5` dove `ts` rappresenta il *timestamp* di inizio classifica.

2 Architettura

L'architettura utilizzata per l'analisi dei dati si compone di molteplici JVM (*Java Virtual Machine*), di diversi *container* Docker e di un *client* Kafka su macchina locale. Sui *container* Docker è istanziato un *cluster* Kafka (*broker* e Zookeeper) per l'*ingestion* delle tuple ed il salvataggio degli *output* dei processamenti, sulle JVM vengono eseguiti i Producer e i Consumer di Kafka, il *framework* per il *data stream processing* Flink e la libreria, anche questa per l'esecuzione di DSP, Kafka Streams. In locale, in fine, viene eseguito un *client* Kafka per la creazione delle topiche necessarie al processamento.

I *container* Docker comunicano tra loro mediante una rete interna creata automaticamente da Docker Compose; per permettere al *client*, al Producer, ai Consumer e al *framework* di raggiungere il *cluster* di *broker* Kafka, è necessario che il sistema operativo *host* consenta di effettuare il *routing* verso indirizzi IP appartenenti alla sottorete di Docker. Questa condizione è verificata se si utilizza Linux ma non se il sistema operativo *host* è macOS (a causa di dettagli implementativi relativi all'applicazione Docker Desktop).

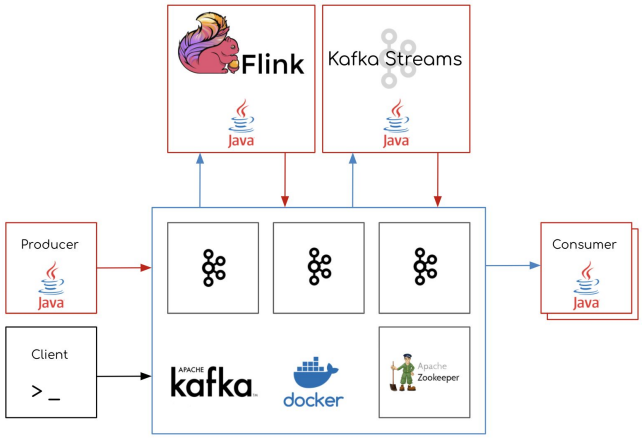

Figura 1: Schema dell'architettura

Di seguito verranno descritte le configurazioni e l'architettura di ogni componente.

2.1 Kafka

Kafka è il sistema di messaggistica di tipo *publish-subscribe* utilizzato per l'*ingestion* di dati nei sistemi di processamento e per l'*export* dei risultati. Il *cluster*, realizzato con Docker Compose, prevede un *container* con Zookeeper, necessario per la coordinazione, e altri tre *container* con la funzione di Kafka *broker*. Sono state create 16 topiche: una per le tuple in *input* a Flink, una per le tuple in *input* a Kafka Streams, sei per l'*output* della prima *query* (giornaliero, settimanale e mensile, rispettivamente per Flink e Kafka Streams), quattro per l'*output* della seconda *query* (giornaliero e settimanale, rispettivamente per Flink e Kafka Streams) e altrettante per quello della terza *query*. Per incrementare la tolleranza ai guasti, ogni Kafka *topic* è impostata per avere un grado di replicazione pari a 2 (una replica *leader* ed una replica *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; in Kafka, infatti, la garanzie di ordinamento sono valide soltanto nell'ambito di una singola partizione.

2.2 Producer

Il Producer si occupa di leggere il *dataset*, mantenuto in un file CSV locale, e di pubblicarne il contenuto presso le topiche di Kafka (*flink-topic* e *kafka-streams-topic*) da cui i *framework* attingono per reperire i dati. La scrittura sulle topiche viene effettuata riga per riga a intervalli di 10 millisecondi (parametro configurabile) al fine di simulare una vera sorgente di dati *real time* accelerata.

Per garantire la corretta esecuzione del processamento su Kafka Streams in base all'*event time*, è stato necessario estrarre la data di occorrenza dell'evento di ogni riga e impostarla come *timestamp* della relativa tupla prima della pubblicazione sulla topica *kafka-streams-topic*.

2.3 Consumer

Sono stati realizzati 14 Consumer (uno per ognuna delle topiche Kafka di *output*) che girano parallelamente in *thread* della JVM, ogni Consumer ha il compito di registrarsi su una topica e leggerne il contenuto. I Consumer si differenziano a seconda del Producer che scrive sul Kafka *topic* a cui si registrano, infatti, quelli relativi a topiche contenenti gli *output* del processamento Flink si occupano di salvare le tuple su file CSV posti nella *directory* Results, mentre, i Consumer di *topic* contenenti gli *output* del processamento Kafka Streams, invece, eseguono la sola stampa su *console* di tali tuple.

2.4 Flink

Flink è il framework di *data stream processing* utilizzato per l'esecuzione delle tre *query* precedentemente descritte.

I dati necessari al processamento sono presi direttamente dalla topica *flink-topic* in Kafka; da questi si ottiene, estraendolo dal campo `Occurred On`, il *timestamp* identificativo dell'evento che viene assegnato come *event time* della tupla e su cui si basa la generazione dei

watermark necessaria per il corretto processamento, anche in caso di *stream out-of-order*.

Il flusso così ottenuto rappresenta lo *stream* che le *query* devono manipolare al fine di calcolare le statistiche richieste; di seguito viene mostrata la topologia che sarà successivamente descritta più nel dettaglio.

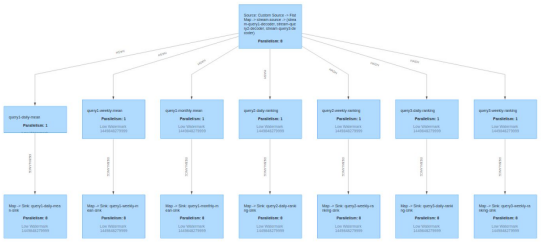

Figura 2: Schema della topologia usata in Flink

2.5 Kafka Streams

Kafka Streams è la libreria di *data stream processing*, che si interfaccia con il *cluster* Kafka, utilizzata per eseguire una valutazione alternativa (in termini di strumento di analisi) delle tre *query* precedentemente descritte.

I dati necessari al processamento sono reperiti dalla topica *kafka-streams-topic* e interpretati, inizialmente, come coppie *key-value* con chiave il *timestamp* dell'evento. A partire da questi dati viene creato lo *stream* con la relativa topologia di processamento che, infine, sarà avviata.

3 Riconoscimento dei ritardi

L'esecuzione della prima e della terza *query* richiede il corretto *parsing* del campo `How Long Delayed` relativo al ritardo accumulato dai bus. Il campo è registrato all'interno del CSV come stringa in linguaggio naturale (non esente da errori di battitura) che identifica un intervallo temporale. La maggior parte delle librerie disponibili *online* per il processamento di date riportate in linguaggio naturale non si aspetta in *input* un intervallo temporale espresso in ore o minuti, di conseguenza si è optato per la scrittura di una libreria specializzata al *parsing* di questo tipo di stringhe. Tale scelta si traduce in maggiore leggerezza del programma (poiché non vengono importate tutte le funzionalità di librerie esterne che si sono rivelate essere spesso sovradimensionate rispetto alle necessità del caso), migliori *performance* e accuratezza (in quanto la libreria è stata creata appositamente per il *parsing* del tipo di stringhe contenute nel *dataset*) e una migliore tolleranza agli errori di battitura.

Ogni stringa, come operazione preliminare, viene pulita da eventuali spazi e simboli e, in seguito, convertita in caratteri minuscoli. Nel caso in cui siano presenti caratteri come "/" o "-", si effettua uno *split* della stringa e si procede alla traduzione di ogni sua parte separatamente, viene poi calcolata la media in minuti dei diversi risultati. Questa operazione garantisce una maggiore tolleranza a intervalli di tempo approssimativi, ad esempio stringhe del tipo "*1/2 hours*" o "*45 min/1 hour*" vengono tradotte rispettivamente in 90 minuti e 52,5 minuti.

Ogni singola stringa (o sua parte) può essere di due tipi: il primo prevede la presenza solo di un valore numerico, il secondo, oltre a tale valore, contiene un'indicazione temporale (minuti o ore). Il risultato del *parsing* di ogni parte è restituito come una tripla contente minuti di ritardo, ore di ritardo e la maggiore unità di misura temporale rilevata all'interno della stringa (tra ore e minuti). Quest'ultimo campo risulta essere fondamentale nel caso di stringhe contenenti molteplici intervalli temporali (separati da "/" o "-"), in tal caso, infatti, si procede al *parsing* di ogni sottostringa da destra verso sinistra propagando di volta in volta la maggiore unità di tempo rilevata.

Questa operazione permette di interpretare correttamente ritardi composti da soli valori numerici (primo tipo) che si affidano alle parti successive per la corretta interpretazione dell'unità temporale; ad esempio, in "1/2 hours" la parte contenente "1" va interpretata come 60 minuti ma tale informazione è racchiusa nella stringa "2 hours", senza propagazione di tale informazione, la lettura del singolo valore, infatti, risulterebbe essere fuorviante. Un'interpretazione come questa viene applicata solo nel caso in cui nella stringa (o nella sua parte) non ci sia indicazione dell'unità temporale; la propagazione, infatti, viene ignorata nel caso in cui la parte sia correttamente formata, ad esempio "45 min/1 hour" considera come unità nella prima stringa i minuti e non le ore, considerate, invece, per la seconda.

Il riconoscimento delle unità si basa sulla ricerca di stringhe inizianti per "m" (nel caso di minuti), per "h" (nel caso di ore) o senza lettere (nel caso di tempi senza unità di misura), questo rende il *parser* maggiormente tollerante ad errori di battitura o a differenti convenzioni per l'espressione dell'unità temporale (ad esempio: "min", "mins", "mt", "mts", "m", "minutos"). Nel caso di traduzione di una intera stringa senza unità di tempo riportata, viene automaticamente considerato il minuto, allo stesso tempo sono ignorate tuple in cui il ritardo è espresso in giorni (si è deciso di considerare questo tipo di ritardo come un *outlier* sulla base dell'analisi delle voci presenti nel dataset) o come data (ad esempio "12 ott"). Il risultato finale dell'operazione di *parsing* è un quantitativo di minuti (non necessariamente

intero) indicante il ritardo medio espresso all'interno della stringa.

4 Finestre temporali user-defined

Per ottenere un processamento temporalmente corretto e che rispettasse le richieste precedentemente riportate, è stato necessario, sia per il processamento con Flink che con Kafka Streams, creare delle finestre temporali *user-defined*.

4.1 Flink

In Flink è stata sviluppata una *tumbling window* mensile che permettesse di suddividere correttamente le tuple a prescindere dal numero di giorni presenti nello specifico mese. Il funzionamento di tale finestra si basa sull'utilizzo di un *assigner* modificato che, sfruttando l'*event time* associato alla tupla per ottenere il mese di riferimento, assegna ogni dato ad una finestra con tempo di inizio impostato al primo giorno del mese in questione (ore 00:00.000) e tempo di fine all'ultimo giorno dello stesso (ore 23:59.999).

4.2 Kafka Streams

In Kafka Streams è stato necessario sviluppare delle *tumbling window* personalizzate per tutte le categorie di finestra necessarie (giornaliere, settimanali e mensili). Tutte queste finestre sono state create in modo che l'inizio della finestra di processamento combaci con la mezzanotte relativa alla *time zone* selezionata; le finestre settimanali sono state allineate ai lunedì mentre quelle mensili al primo giorno del mese.

Anche in questo caso, grazie all'*event time* a loro attribuito, è possibile associare le tuple alla finestra corretta.

5 Query 1 workflow

L'esecuzione della prima *query* inizia con l'estrazione delle informazioni necessarie al processamento: `Occurred On`, `How Long Delayed` e `Boro`; questi dati vengono incapsulati all'interno di una classe provvedendo alla conversione del ritardo da stringa di caratteri al valore numerico in minuti grazie all'utilizzo del *parser* sopra riportato. Eventuali tuple che risultano essere malformate vengono automaticamente ignorate nel processamento.

5.1 Processamento Flink

Il processamento tramite Flink prosegue sfruttando le finestre temporali, in particolare, per quelle giornaliere e settimanali, sono state utilizzate delle *tumbling window* rispettivamente di 1 e 7 giorni, mentre, per quelle mensili, sono state utilizzate le finestre *user-defined* precedentemente riportate (paragrafo 4.1).

Per quanto riguarda le finestre giornaliere e settimanali, poiché, come riportato in [2], le finestre di *default* in Flink sono allineate alla mezzanotte di UTC-0, è stato necessario specificare un *offset* di 4 ore al fine di far coincidere l'inizio delle finestre con la mezzanotte (ore 00:00.000) del fuso orario utilizzato negli *event time*.

Su tutte le finestre viene applicata una funzione di *aggregate* personalizzata che permette, al contrario della *process*, di aggiornare le statistiche della *window* ogni volta che una tupla le viene assegnata; questa accortezza consente di evitare picchi di carico dovuti alla computazione in blocco di tutte le tuple assegnate a una finestra al completamento della stessa. La funzione di aggregazione si occupa di mantenere aggiornati, per ogni quartiere/contea, il tempo di ritardo totale e il numero di disservizi verificatisi, tali informazioni saranno poi combinate per il calcolo della media alla chiusura della finestra. Allo stesso tempo, per poter riportare nei risultati la data corretta di inizio della finestra temporale, è stato necessario utilizzare una *process* adibita all'estrazione di tale informazione dalla *window*. I risultati così ottenuti vengono, quindi, convertiti in stringhe, rispettando il formato presentato nel paragrafo 1.1, e pubblicati sulle relative topiche Kafka.

5.2 Processamento Kafka Streams

Il processamento con Kafka Streams, invece, prosegue con una funzione di *map* che imposta la chiave in modo appropriato (chiavi giornaliere, settimanali o mensili in base al tipo di processamento considerato). La necessità di specificare una chiave differente da quella originaria (il *timestamp* della tupla) è dovuta alla possibilità di applicare, in Kafka Streams, il raggruppamento per finestre soltanto a Keyed Stream e alla generazione di una differente finestra temporale per ogni valore di chiave incontrato.

A questo punto, vengono utilizzate le finestre giornaliere, settimanali e mensili *custom* (riportate nel paragrafo 4.2) su cui, per poter calcolare le statistiche, similmente a prima, si applica una funzione di aggregazione del tutto equivalente a quella usata nel processamento con Flink.

Poiché Kafka Streams non offre molte garanzie sull'ordine di processamento, per poter gestire tuple *out-of-order*, vengono impostati dei *grace period* alle finestre; l'utilizzo di questi valori consente di ritardare la chiusura della finestra temporale attendendo eventuali dati in ritardo. Le tuple che, subendo ulteriori ritardi, arrivano dopo la chiusura della finestra vengono, quindi, soppresse.

Infine, similmente a come accade nel processamento con Flink, i risultati vengono trasformati in stringhe e pubblicati presso le relative topiche Kafka.

6 Query 2 workflow

Il processamento della seconda query inizia, in modo simile a ciò che avviene per la prima, con l'estrazione dalla tupla delle informazioni di interesse per l'analisi: `Occurred On` e `Reason`; anche in questo caso, eventuali tuple malformate rispetto a questi due campi sono automaticamente scartate.

6.1 Processamento Flink

In Flink si procede raccogliendo i dati in delle *tumbling window* di durata pari a 1 e 7 giorni con *offset*, anche qui, di 4 ore. Per poter calcolare e aggiornare efficientemente il *ranking* delle motivazioni di ritardo man mano che le tuple vengono assegnate alla finestra, è stata definita una funzione di aggregazione *custom* che si occupa di riconoscere in quale delle due fasce orarie definite (05:00-11:59 o 12:00-19:00) il dato ricada e di aggiornare la corrispondente classifica parziale dei ritardi in base alla nuova informazione. Al completamento della finestra, viene emesso il *ranking* finale contenente, per ognuna delle due fasce orarie, le tre ragioni di ritardo più frequenti. Al fine di riportare la data di inizio della finestra, è stato necessario l'utilizzo di una *process window* la cui unica funzione, in modo simile a prima, è quella di ottenere tale informazione dalla finestra appena chiusa. I risultati, sia giornalieri che settimanali, quindi, sono convertiti in stringhe del formato indicato nel paragrafo 1.2 per poter essere pubblicate sui *topic* Kafka *flink-output-topic-query2-daily* e *weekly*.

6.2 Processamento Kafka Streams

A causa della caratteristica precedentemente riportata, anche in questo caso, alle tuple vengono cambiate le chiavi che saranno assegnate su base giornaliera o settimanale. I due Keyed Stream vengono quindi raggruppati sfruttando le finestre temporali giornaliere e settimanali descritte prima, su tali finestre vengono applicate le stesse funzioni di aggregazione viste nel caso di Flink. Anche qui, come accade per la prima *query*, è stato impostato il *grace period* e si è deciso di sopprimere le eventuali tuple arrivate in ulteriore ritardo. I risultati, infine, vengono pubblicati sui topic Kafka *kafka-streams-output-topic-query2-daily* e *weekly* dopo essere stati convertiti in stringhe.

7 Query 3 workflow

Il processamento della terza *query* inizia, similmente alle prime due, con l'estrazione dalla tupla dei dati di interesse: `Occurred On`, `How Long Delayed`, `Bus Company Name` e `Reason`; anche in questo caso, come nei precedenti, eventuali tuple malformate rispetto a questi campi sono automaticamente scartate.

7.1 Processamento Flink

In Flink, come per la seconda *query*, si procede raccogliendo i dati in delle *tumbling window* della durata di 1 e 7 giorni e, anche in questo caso, specificando un *offset* di 4 ore. Al fine di calcolare efficientemente la classifica delle compagnie con punteggio di disservizio più alto è stata usata una funzione di aggregazione *custom* che si occupa di aggiornare, all'arrivo di ogni nuova tupla, lo *score* di disservizio della compagnia a cui questa fa riferimento. Al completamento della finestra, viene restituita la classifica delle cinque compagnie con punteggio più alto. Per riportare correttamente la data di inizio della finestra, come per le altre due *query*, è stata utilizzata una *process window*. I risultati ottenuti, sia giornalieri che settimanali, sono trasformati in stringhe che rispettano il formato descritto nel paragrafo 1.3 e pubblicati presso i corrispondenti *topic* Kafka.

7.2 Processamento Kafka Streams

L'esecuzione della terza *query* sfruttando la libreria Kafka Streams prevede, come per le altre due, la conversione delle chiavi o su base giornaliera o settimanale. A questo punto si prosegue con il raggruppamento dei due Keyed Stream in finestre temporali giornaliere e settimanali impostando, anche in questo caso, il *grace period* e sopprimendo eventuali tuple che arrivano oltre lo scadere di questo intervallo temporale. Su queste finestre vengono applicate delle funzioni di aggregazione analoghe a quelle usate nell'esecuzione con Flink e, al completamento di una finestra, i risultati della computazione della stessa vengono convertiti in stringhe e pubblicati sui *topic* Kafka.

8 Benchmark

La valutazione delle prestazioni è stata effettuata su sistema operativo Linux Ubuntu 20.04 virtualizzato con, a disposizione, 8 GB di RAM e 8 *core* (8 *core* di 16, Intel i9 3,6 GHz - 5,0 GHz) sia utilizzando gli strumenti messi a disposizione da Flink e Kafka Streams che sperimentalmente; le metriche messe a disposizione da Kafka Streams, tuttavia, non vengono riportate per operatore e, quindi, ai fini di un confronto con il *framework* di *data stream processing*, non sono state ritenute indicative, pertanto non sono state inserite nella presente relazione.

Per poter eseguire una valutazione sperimentale dei *benchmark* di ogni operatore, è stata utilizzata, per ognuno

di essi, una struttura tenente conto sia del numero di tuple processate che del tempo impiegato; l'accesso a tale struttura è effettuato, per evitare inconsistenze nel caso di aggiornamenti simultanei da parte di operatori replicati, mediante l'utilizzo di metodi `synchronized`.

Sia nell'utilizzo di Kafka Streams che di Flink, l'*update* delle statistiche viene effettuato prima della pubblicazione del risultato sul relativo Kafka *topic*, i tempi di pubblicazione, quindi, non sono stati considerati; nel secondo caso, in particolare, è stato necessario l'utilizzo di un Sink alternativo.

La valutazione delle prestazioni è stata eseguita utilizzando la sorgente di dati che pubblica una tupla ogni 10 millisecondi sui *topic* di input con semantiche *exactly-once* per la comunicazione.

Operatore	**Performance di Flink**		
	Throughput Flink UI (tuple/sec)	**Throughput sperimentale (tuple/sec)**	**Latenza sperimentale (sec/tupla)**
Query 1 daily	0,338	0,285	3,508
Query 1 weekly	0,071	0,062	16,05
Query 1 monthly	0,016	0,016	60,8
Query 2 daily	0,344	0,303	3,35
Query 2 weekly	0,071	0,065	15,7
Query 3 daily	0,338	0,283	3,575
Query 3 weekly	0,071	0,064	15,83

Tabella 1: Benchmark di Flink

Operatore	**Performance di Kafka Streams**	
	Throughput sperimentale (tuple/sec)	**Latenza sperimentale (sec/tupla)**
Query 1 daily	0,344	3,04
Query 1 weekly	0,064	15,8
Query 1 monthly	0,019	52,05
Query 2 daily	0,332	3,01
Query 2 weekly	0,066	15,2
Query 3 daily	0,321	3,120
Query 3 weekly	0,066	15,23

Tabella 2: Benchmark di Kafka Streams

Confrontando queste due alternative per il *data stream processing* soltanto da un punto di vista di *throughput* e latenza, Kafka Streams sembrerebbe essere la soluzione migliore, seppur di poco. In realtà, per una valutazione completa, andrebbero considerate anche le garanzie sul processamento che i due differenti approcci comprendono: infatti, viste le garanzie dell'ordine di processamento offerte in Flink mediante i *watermark*, l'accuratezza della computazione offerta dal *framework* di *data stream processing* risulta essere, a parità di *input*, superiore di quella della libreria Kafka Streams che, in caso di ritardi troppo elevati, esegue il *drop* delle tuple.

REFERENCES

[1] http://www.ce.uniroma2.it/courses/sabd1920/projects/prj2_dataset.zip

[2] https://ci.apache.org/projects/flink/flink-docs-stable/dev/stream/operators/windows.html