Analisi Stream di Dati Finanziari con Apache Flink

SIMONE NICOSANTI, 0334319, University of Rome Tor Vergata, Italy

Questo documento ha lo scopo di presentare l'architettura di sistema, le strategie e i risultati sperimentali di un'analisi di dati finanziari in modalità stream con Apache Flink

1 INTRODUZIONE

Lo scopo del progetto è quello di fare stream processing di dati finanziari presi da un dataset ottenuto dalla Fintech Infront Financial Tecnology usando Apache Flink come framework di elaborazione. Il dataset riguarda scambi di strumenti finanziari su tre mercati europei nella settimana dall'8 al 14 novembre 2021. Allo scopo del progetto è stata considerata una versione ridotta del dataset: questa versione consiste di circa 4 milioni di eventi (a fronte delle 289 milioni del dataset originario) che riguardano circa 500 strumenti tra azioni ed indici scambiati sui mercati europei di Parigi (FR), Amsterdam (NL) e Francoforte/Xetra (ETR).

2 ARCHITETTURA DI SISTEMA

Per quanto riguarda l'architettura di sistema, abbiamo deciso di emularla su singolo nodo usando Docker Compose. Le parti principali del sistema sono le seguenti:

- Producer. Non disponendo di dispositivi che potessore emettere gli eventi, volendo simulare al meglio un contesto di elaborazione streaming, abbiamo introdotto un nodo Producer, che si occupa di effettuare il replay del dataset originario su un topic Kafka. Il file csv originale viene scaricato dal web usando il comando *curl*. Il producer è stato scritto in python usando la libreria di python *kafka-python*.
- Apache Kafka. Apache kafka è stato usato per la scrittura dei record del dataset originario. I record vengono scritti su un topic *flink-topic* che viene creato automaticamente quando si tenta la scrittura della prima tupla. Allo stesso modo Kafka è stato usato per la scrittura dei risultati delle query, creando un topic diverso per ogni query e per ogni fascia oraria considerata.
- Apache Flink. Apache Flink è stato usato come framework per l'elaborazione stream dei dati. Il codice di esecuzione delle query è stato scritto in python usando la libreria *pyflink* di python. Essendo il connettore kafka non fornito nella libreria, il jar viene scaricato dal web usando il comando *curl* alla creazione dell'immagine di container tramite dockerfile e tale jar viene aggiundo all'env alla creazione dell'execution environment di Flink.
- Consumer. Usato per leggere da Kafka i risulati delle queries e per salvarli in un file csv. Il consumer si registra sui topic kafka corrispondenti alle singole queries: quando una tupla viene inserita il consumer la riceve e la salva in un file csv specifico per la query avente lo stesso nome del topic di origine.
- Prometheus. Usato per raccogliere i dati di througput e latenza di flink. Per usarlo è stato esposto su Flink un *metrics.reporter* a cui Prometheus chiede periodicamente questi valori.
- Grafana. Usato per graficare i dati raccolti da Prometheus in un modo più accessibile all'utente

3 REPLAY DEL DATASET E PRE ELABORABORAZIONE

Del dataset originario, al fine dell'esecuzione delle queries di interesse, erano di interesse solo i seguenti campi:

Author's address: Simone Nicosanti, 0334319, snicosanti@gmail.com, simone.nicosanti.01@students.uniroma2.eu, University of Rome Tor Vergata, Rome, Italy.

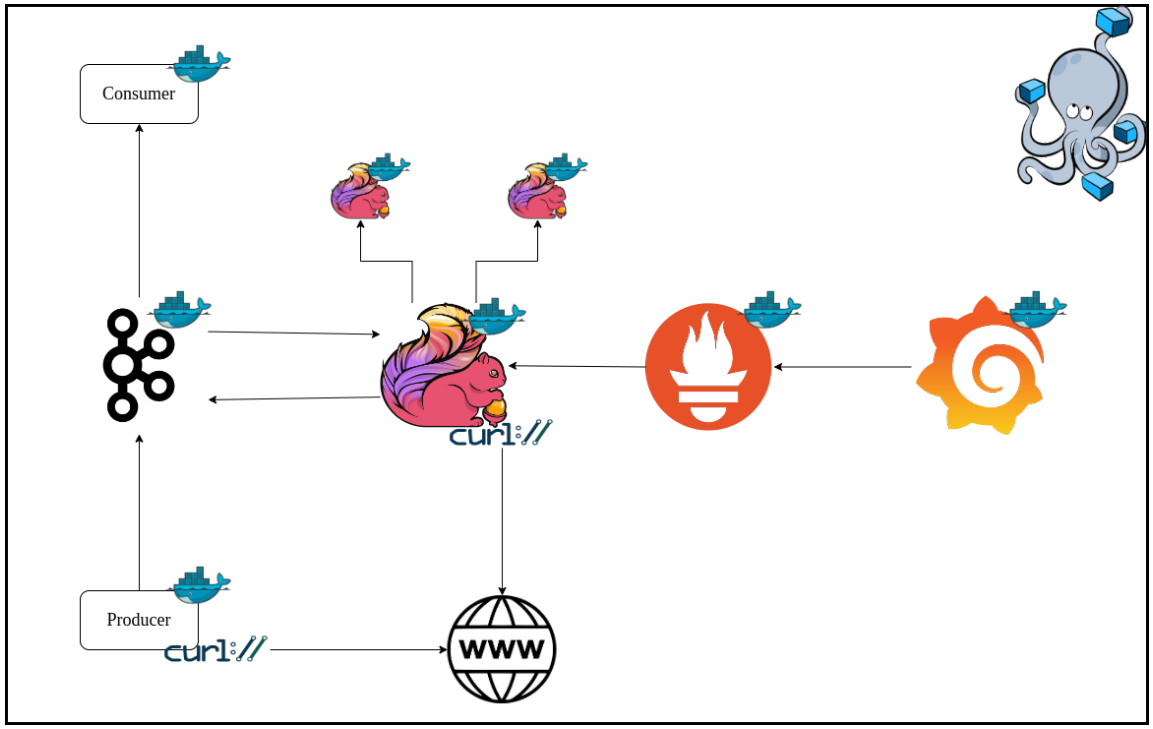

Fig. 1. System Architecture

- ID
- SecType
- Date e Time
- Last
- TradingDate e TradingTime

Sono stati letti solo questi campi, per evitare di appesantire il framework di elaborazione. Le tuple lette sono state ordinate per Date e Time. Essendo presente nel dataset originario tuple aventi stessi valori di Date e Time, per simulare l'invio contemporaneo di questi dati sul topic, è stato cambiato il parametro *lingerMs* del producer kafka per fare in modo che l'invio delle tuple avvenisse o allo scadere del tempo indicato oppure se forzato con una flush: le tuple aventi stessa Date e Time quindi vengono inviate sul producer che le bufferizza fino a quando non viene invocata la flush al cambio di uno dei due indicatori temporali. Le tuple sono convertite in stringhe in formato JSON e poi riconvertite lato Flink. Per permettere il trigger delle ultime finestre delle queries, sono state aggiunte delle tuple "fittizie" con valori nulli tranne ID e SecType e con TradingDate e TradingTime poste rispettivamente al *20-11-2021* e *12:00:00.000*: è stata generata una tupla di questo tipo per ogni ID presente nel dataset.

I dati scritti in Kafka vengono poi letti da Flink tramite un *KafkaSource* di Flink. Tramite una *Map* le stringhe JSON sono rinconvertite in tuple. Dopo la lettura dei dati questi sono stati filtrati:

- Considerando che le queries sono basate sul tempo, abbiamo deciso di rimuovere tuple con *TradingDate* e *TradingTime* nulli.

- Da un'analisi del dataset si è visto che tutte le tuple con *TradingTime* pari a *00:00:00.000* presentavano valore del campo *Last* nullo, abbiamo deciso di rimuovere queste tuple, considerandole come tuple di reset del sistema (anche in considerazione del fatto che in un contesto reale la borsa chiude di pomeriggio o di sera).
- Da un'analisi del dataset si è visto che erano presenti delle tuple che, a parità di *ID*, *TradingDate* e *TradingTime*, presentavano diversi valori di Last: queste tuple sono state lasciate invariate, considerandole come se fossero state generate da sorgenti diverse che hanno registrato valori diversi nello stesso momento

Terminato il filtraggio, i campi *TradingDate* e *TradingTime* sono stati convertiti in un timestamp usando la libreria *datetime* di python e il valore ottenuto è stato moltiplicato per 1000 per passare al timestamp espresso in millisecondi, che è quello usato da Flink di default. Le tuple risultanti da queste operazioni sono del tipo: (ID, SecType, Last, Timestamp)

4 QUERY 1

Query Per le azioni (campo SecType con valore pari a E) scambiate sui mercati di Parigi (FR) che iniziano per "G", calcolare il numero di eventi ed il valor medio del prezzo di vendita (campo Last). Indicare nel risultato anche il timestamp dell'inizio della finestra temporale. Calcolare la query sulle seguenti finestre temporali:

- 1 ora
- 1 giorno
- Dall'inizio del dataset

Per quanto riguarda la query 1 si è proceduto in questo modo:

(1) Filtraggio per ID e SecType
(2) Assegnazione del timestamp
(3) Map ad una tupla formata da (ID, Last, 1)
(4) KeyBy per ID
(5) Window sulla finestra temporale: in particolare è stata usata una finestra di tipo Tumbling sulle durate indicate. Per l'inizio del dataset, considerato che le tuple fittizie hanno *TradingDate* al *20-11-2021* è stata considerata una finestra di durata di 7 giorni.
(6) Reduce passando ad una tupla di tipo (windowStart, ID, total, count) per ogni ID. L'inizio della finestra è stato preso usando una ProcessWindowFunction accoppiata con la Reduce: questo permette di ottenere, emessa la tupla di output dalla finestra, le informazioni riguardanti la finestra usata e, nello specifico, di prendere l'inizio della finestra.
(7) Map per passare ad una tupla di tipo (windowStart, ID, avg, count)
(8) Map per passare dalla tupla ad una stringa in formato json
(9) SinkTo su Kafka per salvare il risultato della query

5 QUERY 2

Query Calcolare la classifica aggiornata in tempo reale delle 5 azioni (di qualsiasi mercato) che registrano la più alta variazione del prezzo di vendita calcolato nella finestra temporale indicata di seguito e delle 5 azioni (di qualsiasi mercato) con la variazione del prezzo di vendita più bassa. Indicare nella query il timestamp relativo all'inizio della finestra. Calcolare la query sulle seguenti finestre temporali:

- 30 minuti

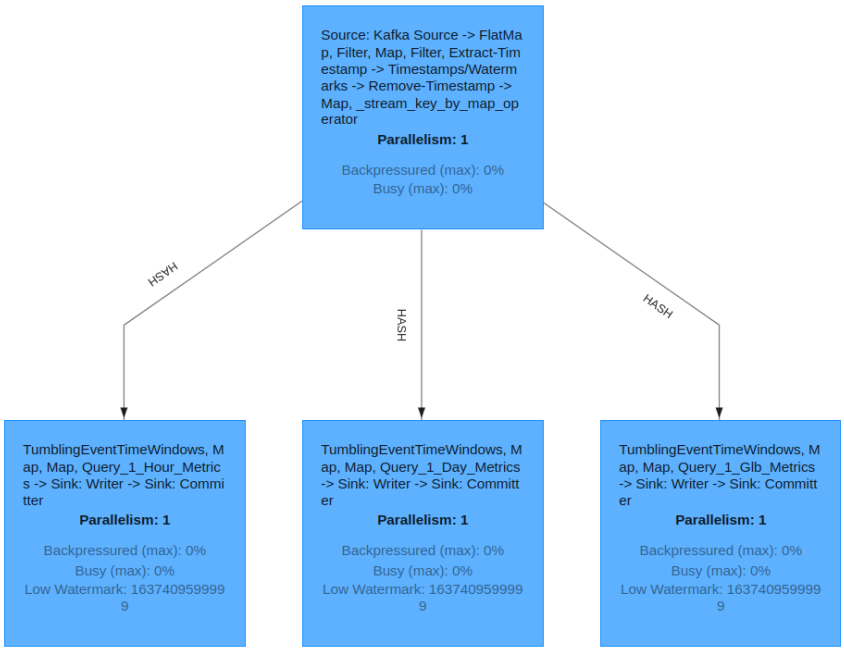

Fig. 2. Schema Query 1

- 1 ora
- 1 giorno

I passi fatti per l'implementazione della query sono i seguenti:

(1) Assegnazione del timestamp
(2) Map per passare ad una tupla di tipo (ID, timestamp, last, timestamp, last)
(3) KeyBy ID per trovare le variazioni per il singolo ID
(4) Window per travare le variazioni nella finestra temporale specifica: si è usata una TumblingWindow per specificare la finestra
(5) Reduce accoppiata con una ProcessWindow. La ProcessWindow ci permette, dopo aver ottenuto il risulato della Reduce, di ottenere il timestamp associato all'inizio della finestra. Otteniamo tuple di tipo (windowStart, ID, minTime, minLast, maxTime, maxLast), che ci dice, nella fascia della finestra il primo e ultimo valore. La reduce opera su due tuple del tipo sopra in questo modo:
    - Cerca il minimo tra i due minTime e il massimo tra i due maxTime
    - Se i minTime sono uguali, si prende come last1 risultante il minimo dei last1, altrimenti si prende quello associato al minTime più piccolo.
    - Opera in maniera duale sui maxTime. Questo è stato fatto per tenere conto di quei casi in cui a parità di ID, TradingDate e TradingTime vi sono prezzi diversi: stiamo quindi considerando la massima variazione di prezzo che ci può essere stata per quello strumento anche laddove questa variazione si fosse verificata alla stessa ora.

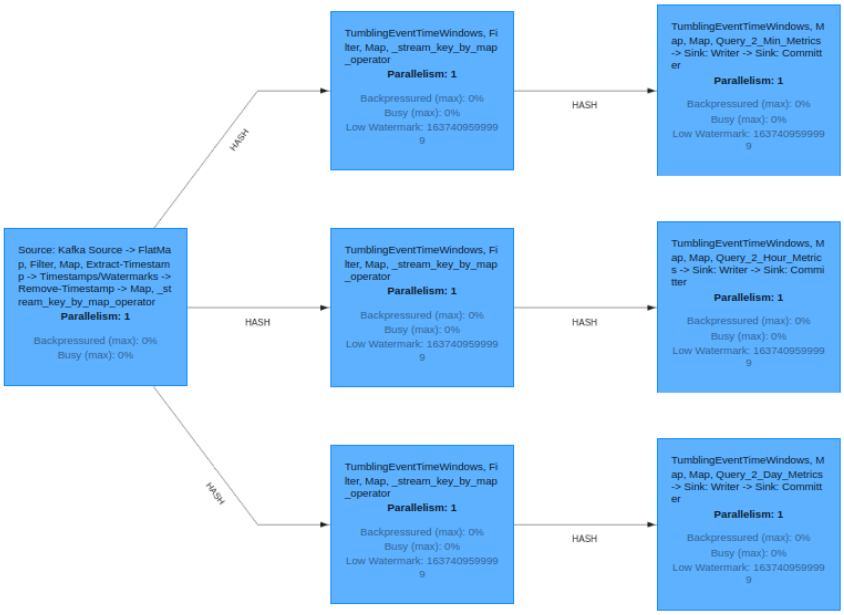

Fig. 3. Query 2

(6) Filter per considerare solo tuple aventi minTime e maxTime diversi oppure, a parità di questi, considerare tuple aventi last1 e last2 diversi: se, a parità di timestamp, anche i valori dei last sono uguali significa che il prezzo dello strumento non ha subito variazioni.

(7) Map per ottenere, per ogni ID, la variazione avuta in una certa finestra temporale, ottenendo tuple di tipo (windowStart, ID, variation) sottraendo i campi *maxLast* e *minLast*

(8) WindowAll sulla stessa finestra temporale usata nella precedente window: questo significa che le tuple della stessa finestra oraria vengono elaborate insieme fino a che non arrivano tuple appartenenti alla fascia oraria successiva.

(9) Aggregate sulla window. La aggregate opera nel seguente modo, permettendoci di ottenere una tupla del tipo (windowStart, rankingList)

- Crea un accumulatore dato da (0, emptyList), dove 0 indica il valore del timestamp
- Quando arriva una nuova tupla aggiunge la coppia (variation, ID) alla lista: la lista viene poi ordinata per variazione e se la sua lunghezza è minore o uguale a 10 viene lasciata invariata, altrimenti ne vengono estratti i primi 5 e gli ultimi 5. Questo ci permette di calcolare la classifica in modo incrementale, senza mantenere tutte le tuple in memoria

(10) Map per passare da una tupla di tipo (windowStart, rankingList) ad una tupla di tipo (windowStart, ID1, Var1, ID2, Var2, ..., ID10, Var10)

(11) Map per passare ad una stringa in formato json

(12) SinkTo su Kafka per salvare i risultati della query

6 QUERY 3

Query Dopo aver calcolato la variazione di prezzo di vendita delle azioni scambiate sui diversi mercati per la finestra temporale di seguito indicata, raggruppare le azioni per mercato (ETR, FR e NL) e calcolare il 25-esimo, 50-esimo, 75-esimo percentile della variazione del prezzo di vendita per ciascun mercato. Indicare nella query il timestamp relativo all'inizio della finestra. Calcolare la query sulle seguenti finestre temporali:

- 30 minuti
- 1 ora
- 1 giorno

I passi implementativi di questa query sono i seguenti:

(1) Dovendo anche qui trovare le variazioni per ogni strumento, i passi da *1* a *7* sono uguali a quelli della query 2. Otteniamo quindi delle tuple del tipo (windowStart, ID, variation)
(2) Map per estrarre dall'ID il mercato di appartenenza, ottenendo tuple del tipo (windowStart, market, variation)
(3) KeyBy per mercato e Window per calcolare i valori dei percentili per ogni mercato nella finestra oraria data: la Window è fatta sulla stessa finestra temporale usata in precedenza e quindi valori appartenenti alla stessa finestra temporale saranno elaborati insieme
(4) Aggregate usando una AggregateFunction custom, ottenendo tuple del tipo (windowStart, Market, 25-perc, 50-perc, 75-perc). Questa AggregateFuncion, chiamata *TDigestComputation* sfrutta internamente la libreria python *tdigest* per il calcolo dei percentili in modo incrementale senza aspettare l'arrivo di tutte quante le tuple. In questo caso si è scelto il tdigest per il calcolo dei percentili perché la struttura tdigest supporta la somma ed essendo la somma associativa e commutativa può essere usata per la Aggregate di Flink. Inizialmente si era usata invece la libreria *psquare* che invece implementa l'algoritmo del p^2 per il calcolo del percentile; tuttavia il p^2 non è sommabile e quindi si era rivelato necessaria l'implementazione di una *KeyedProcessFunction* con chiave sull'inizio della finestra e timer impostato al valore di inizio della finestra più la sua durata.
(5) Map per mappare il risultato ad una stringa in formato json
(6) SinkTo su topic Kafka per salvare il risultato della query

7 EMPIRICAL RESULTS

Le simulazioni sono state eseguite su un unico nodo usando Docker Compose. La configurazione di Flink usata per le simulazioni è la seguente:

- Parallelism = 1
- Numero di TaskManager = 1
- Numero di TaskSlot = 10

Le valutazioni sono state fatte considerando due metriche:

- Throughput. Per il Throughput è stata costruita una metrica custom su Flink usando una *RichMapFunction*. Questa Map è stata inserita prima del Sink su Kafka, quindi possiamo considerare con una buona approssimazione il througput della query come il througput di questo operatore. La metrica è stata calcolata prendendo prima di tutto il tempo di creazione dell'operatore; quando una tupla passa per l'operatore, viene preso l'istante attuale, si calcola la differenza con l'istante di creazione dell'operatore, si incrementa il numero di tuple passate fino a

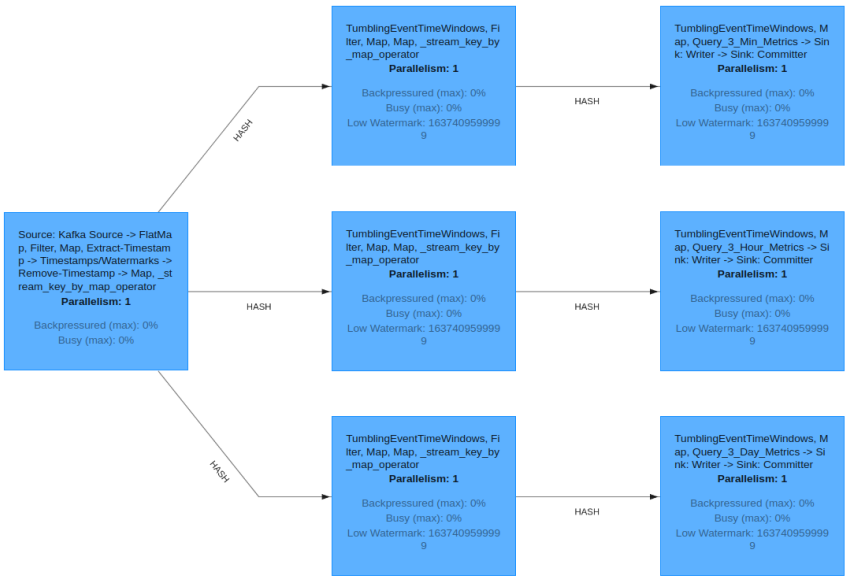

Fig. 4. Query 3

quel momento per l'operatore e si dividono i due valori: quello che otteniamo quindi è una stima dell'evoluzione del throughput nel tempo.

- Latenza. Per la latenza è stata usata la metrica di latenza fornita da Flink stesso, attivando il *LatencyTraking*. Usare questa metrica di Flink ha sollevato tuttavia i seguenti problemi:
  - Si tratta di una metrica Flink di tipo *Histogram* e come tale per essa vengono forniti soltanto i quantili. Per fare una stima per eccesso si è deciso di considerare il quantile 0.999, considerandolo una buona approssimazione della massima latenza
  - I valori della latenza non vengono forniti solo end-to-end, ma per il passaggio lungo tutti gli operatori della query. Questi operatori vengono identificati nella metrica tramite il loro ID che però è un hash calcolato da Flink all'avvio e quindi difficile da identificare per un utente all'atto della richiesta del valore: questo ha quindi portato all'impossibilità di identificare il Sink tra i vari operatori. Il ragionamento che si è seguito quindi è stato il seguente: se un operatore si trova tra la sorgente ed il Sink, la latenza tra sorgente e operatore è per forza minore di quella tra sorgente e Sink, quindi si è ritenuto congruo scegliere tra i valori restituiti il massimo.

  In conclusione la statistica presa per la latenza è il massimo dei 0.999 quantili della latenza dei vari operatori.

La modalità di esecuzione delle queries è stata la seguente:

- Ogni query eseguita singolarmente
- Data una query, la query su fasce orarie diverse è eseguita separatamente in Job diversi, per monitorare la latenza che si ha con la singola fascia oraria.
- Query esegita contestualmente alla produzione dei dati da parte del producer

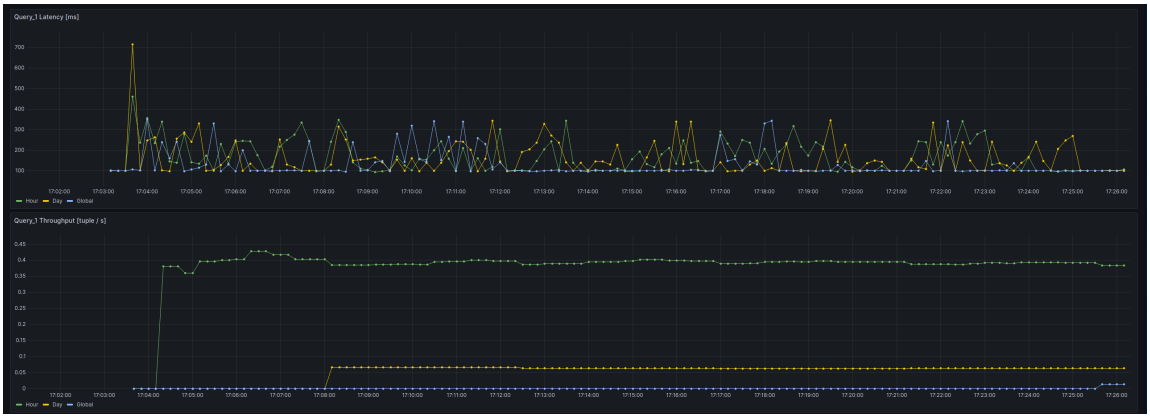

Fig. 5. Query 1 Metrics

Le metriche vengono monitorate da Prometheus ogni secondo e poi graficate su Grafana in modalità real time con aggiornamento ogni 10 secondi.

Per quanto riguarda il throughput, possiamo notare come in generale il throughput della query sia inversamente proporzionale alla dimensione della finestra: questo viene dal fatto che essendo la finestra più piccola i valori saranno elaborati più velocemente, quindi arriveranno più velocemente alla fine del flusso di esecuzione. Per quanto riguarda le query 2 e 3, possiamo notare degli andamenti molto simili per il throughput, in quanto la sequenza di operazioni fatta nei due casi è molto simile. Notiamo però come la query 2 presenti un throughput minore in generale: questo è probabilmente dovuto alle operazioni di ordinamento che sono fatte nell'esecuzione della query 2, mentre nella query 3 abbiamo l'uso del t-digest che è una struttura dati che opera più velocemente.

Per quanto riguarda la latenza, possiamo notare come quella delle finestre temporali maggiori tenda ad essere più bassa: questo perché il *LatencyTracker* di Flink può "scavalcare" le finestre e gli operatori, a meno che questi operatori non abbiano delle tuple accodate; quando la finestra è fatta su finestre temporali grandi molto tempo sarà speso ad attendere l'accumulo delle tuple nelle finestre e i tracker possono passare avanti senza problemi. Vediamo infatti come nelle query a fascia globale e giornaliera sia tendenzialmente più bassa rispetto alle altre finestre della stessa query: per quanto riguarda i picchi di queste fasce, questi li possiamo attribuire da un lato alla condivisione delle risorse sul singolo nodo, dall'altro agli operatori che vengono eseguiti prima dell'esecuzione della query stessa e che preparano le tuple per l'elaborazione.

Per quanto riguarda la query 1, notiamo come i valori di latenza tra la fascia oraria di un'ora e di un giorno siano abbastanza simili e questo può essere dovuto alla natura della query stessa. La query chiede di trovare il valor medio del prezzo di vendita per degli strumenti finanziari con certe caratteristiche: considerato che i dati provengono da un dataset reale, verosimilmente gli strumenti che sono considerati non sono diversi se consideriamo fasce orarie diverse e poiché il calcolo è fatto in maniera incrementale nella finestra, gli operatori successivi si troveranno ad elaborare più o meno lo stesso numero di tuple e quindi un marker che passa troverà circa la stessa coda sugli operatori successivi.

Fig. 6. Query 2 Metrics

Fig. 7. Query 3 Metrics

Per quanto riguarda la latenza delle query 2 e 3, notiamo anche qui andamenti molto simili per lo stesso motivo di prima. Anche qui le latenze con fasce orarie di 30 minuti e un'ora sono molto simili: questo è probabilmente dovuto sia al fatto che le query sono uguali nella prima parte dell'eleborazione, cioè nella ricerca delle variazioni, sia al fatto che il numero di strumenti che vengono scambiati in un ora non è troppo diverso rispetto al numero di strumenti che sono scambiati in 30 minuti, quindi gli operatori successivi elaboreranno più o meno lo stesso numero di tuple. Notiamo anche che in entrambi i casi ci sono dei picchi discendenti: questi potrebbero corrispondere alle latenze in cui abbiamo il cambio del giorno e il flusso di esecuzione si svuota parzialmente.

Vediamo infine come al termine dell'esecuzione delle query, la latenza per le query eseguite con tutte le finestre si stabilizzi intorno ad un certo valore: questo è il tempo impiegato dal *LatencyMarker* ad attraversare il flusso d'esecuzione nel suo complesso a sistema vuoto.