Sistemi e Architetture per Big Data - AA 2021/2022
Progetto2: Analisi in tempo reale di dati ambientali con Flink

Martina De Maio
Corso di laurea magistrale in Ingegneria Informatica
Università degli studi di Roma "Tor Vergata"
martina.demaio@alumni.uniroma2.eu

***Abstract*—La seguente trattazione descrive l'implementazione di una soluzione per una serie di query su un dataset relativo a dati ambientali, provenienti da sensori messi a disposizione da Sensor.Community[1], una rete di sensori globale che ha l'obiettivo di fornire open data di tipo ambientale. La soluzione proposta è stata sviluppata utilizzando il framework di data stream processing Apache Flink e Apache Kafka come sistema di data acquisition e ingestion .**

1. Introduzione

Lo scopo del progetto è effettuare un'analisi in tempo reale sullo streaming di dati proventienti da sensori di tipo BMP180, contenuti all'interno di un dataset relativo a Maggio 2022 che contiene, in particolare, i valori di temperatura, altitudine e pressione misurati da tali sensori.

Il dataset utilizzato è `2022-05_bmp180`, disponibile in formato CSV all'URL [2].

Le 3 query a cui rispondere sono le seguenti:

1.1. Query1

Per i sensori aventi id minore di 10000 (campo `sensor_id`), calcolare il numero delle misurazioni e il valore medio di temperatura.
Tale query è stata calcolata sulle finestre temporali:

- 1 ora (event time),
- 1 settimana (event time),
- dall'inizio del dataset.

1.2. Query2

Calcolare la classifica aggiornata in tempo reale delle 5 località (facendo riferimento al campo `location`) aventi temperatura media più alta e delle 5 località aventi temperature media più bassa.
Tale query è stata calcolata sulle finestre temporali:

- 1 ora (event time),
- 1 giorno (event time),
- 1 settimana.

1.3. Query3

Ai fini di tale query si limita l'area geografica d'interesse alle coordinate di latitudine e longitudine (38°, 2°) e (58°, 30°). Si suppone inoltre che l'area considerata venga divisa in una griglia di dimensione 4x4 e con celle rettangolari, identificando ogni cella da sinistra a destra e dall'alto verso il basso con il nome `cell_X`, dove X è un identificativo da 0 a 15. Ad esempio, la posizione avente coordinate (38°, 2°) si trova in cell_0, mentre la posizione avente coordinate (58°, 30°) si trova in cell_15. Fornire, per ogni cella, la temperatura media e la temperatura mediana (ovvero il 50 percentile), considerando i valori emessi dai sensori all'interno della cella.
Tale query è stata calcolata sulle finestre temporali:

- 1 ora (event time),
- 1 giorno (event time),
- 1 settimana.

2. Architettura

La piattaforma utilizzata per il processamento delle queries è un nodo standalone su cui è stato avviato un cluster Flink per il processamento streaming e un cluster Kafka come layer di messaggistica per fare ingestion dei dati. Si è inoltre utilizzato Zookeeper, servizio centralizzato per la configurazione e sincronizzazione di servizi distribuiti, in questo caso per il funzionamento di Kafka.
Per tale configurazione è stato utilizzato Docker Compose, utile a definire ed eseguire applicazioni multi-container di tipo Docker. Tramite un file YAML, infatti, sono stati configurati i servizi dell'applicativo, che consistono in un'immagine che definisce il container per il master di Flink, una per i diversi worker, una per Zookeeper ed infine l'immagine del Kafka broker.

A tale scopo è stato utilizzato **Docker Compose**, che consente di definire e condividere applicazioni Docker multi-container utilizzando un file yaml denominato `docker-compose.yml`, nel quale si specifica la configurazione dei servizi della propria applicazione, e con un solo comando si creano e avviano tutti i servizi a partire dalla configurazione specificata. Si utilizza per

creare container multipli all'interno di uno stesso host e coordinare l'esecuzione di più container in esecuzione su una singola macchina. Nel docker-compose.yaml sono presenti le immagini dei container da avviare, e, in particolare, per Flink sono stati impostati 1 job manager (nodo master) e uno scaling pari a 3 per il task manager (nodo worker). I files necessari al deployment dell'applicazione si trovano nella repository GitHub https://github.com/martina97/sabd_project2 nella cartella **"docker"**, che contiene il file `dockercompose. yml` e lo script `"start-docker.sh"` per avviare i cluster.

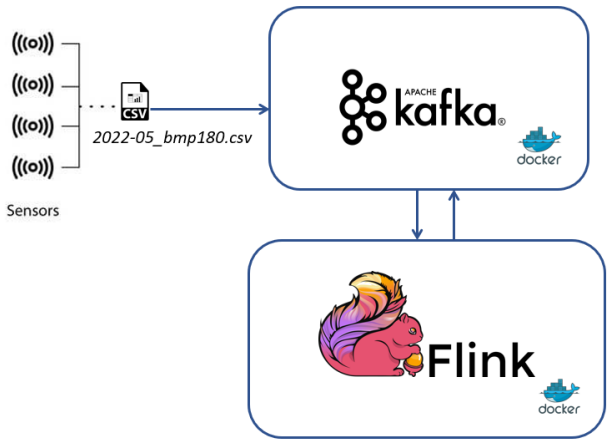


Figure 1. Architettura utilizzata.

2.1. Apache Kafka

Apache Kafka è una piattaforma open source e distribuita di streaming di eventi di tipo **publish/subscribe** sviluppata dalla Apache Software Foundation. I client producono o consumano eventi direttamente da/verso un cluster di broker, che leggono/scrivono eventi in modo duraturo nel file system locale sottostante e replicano automaticamente gli eventi in modo sincrono o asincrono all'interno del cluster per la tolleranza ai guasti e l'elevata disponibilità. Kafka mantiene feed di messaggi in categorie chiamate **topic**: i produttori pubblicano (publish) messaggi su un topic Kafka, mentre i consumatori si iscrivono (subscribe) ai topic ed elaborano i messaggi pubblicati. Cluster Kafka: log di commit distribuito e replicato di dati su server noti come broker, i quali si affidano ad **Apache Zookeeper** per il coordinamento.

2.1.1. Producer Kafka. Il producer si occupa di leggere il dataset e di pubblicarne il contenuto presso il topic "***source***" di Kafka, sottoforma di evento con il relativo **timestamp**. Per simulare una vera sorgente di dati real time si utilizia un **delay** dell'invio dei singoli messaggi proporzionale all'event time, in modo da simulare al meglio un contesto reale. Per garantire la corretta esecuzione del processamento in base all'event time si è estratta la data di occorrenza dell'evento di ogni riga e la si è impostata come timestamp della relativa tupla prima di pubblicarla sul topic "*source*".

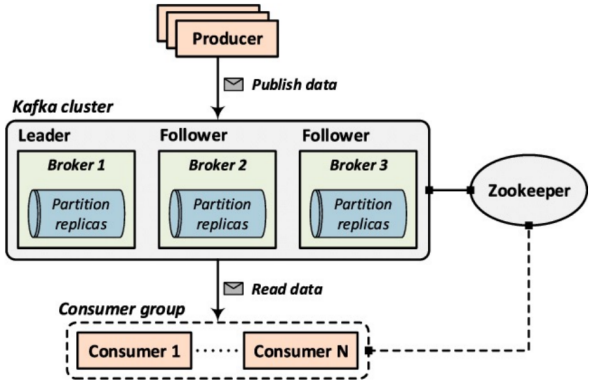


Figure 2. Architettura Kafka.

2.1.2. Consumer Kafka. Il consumer si registra al topic "*source*" e ne legge il contenuto. In particolare, ogni query ha un source in comune, che è il ***FlinkKafkaConsumer***. Dopo il processamento, l'output del flusso viene inviato, tramite un ***FlinkKafkaProducer***, al topic "***results***" creato appositamente su Kafka, e il consumer genera un file in formato csv contenente i risultati.

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

2.2. Apache Flink

Apache Flink è sistema di processamentro distribuito per computazioni stateful su stream di dati illimitati (unbounded) e limitati (bounded), utilizzato per l'esecuzione delle 3 query descritte precedentemente. Una volta letti i dati dal topic Kafka, essi vengono processati tramite opportune operazioni e infine restituiti a Kafka.

Le applicazioni Stateful Flink sono ottimizzate per un Local Access State. Lo stato dell'attività viene sempre mantenuto in memoria o, se la dimensione dello stato supera la memoria disponibile, in strutture di dati su disco efficienti nell'accesso. Pertanto, le attività eseguono tutti i calcoli accedendo allo stato locale, spesso in memoria, producendo latenze di elaborazione molto basse. Flink garantisce consistenza dello stato di tipo exactly-once in caso di guasti controllando periodicamente e in modo asincrono lo stato locale nello storage durevole.

3. Descrizione dell'implementazione

L'applicazione è stata scritta nel linguaggio di programmazione Java ed è stata gestita utilizzando Apache Maven,

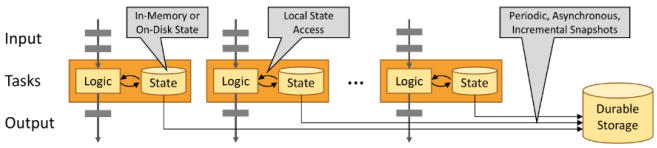


Figure 3. Tolleranza ai guasti in Flink

in modo da facilitare il download automatico delle librerie necessarie e la risoluzione di eventuali dipendenze. Il codice sorgente per rispondere alle 3 query, disponibile nella repository GitHub https://github.com/martina97/sabd_project2, si trova nella cartella *"src/main/java"* ed è suddiviso in tre packages:

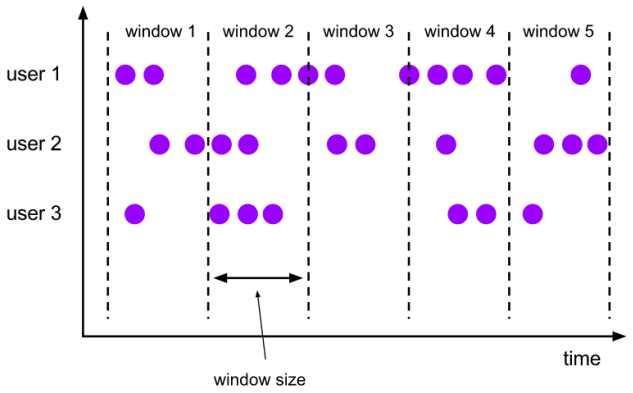
1) ***"kafka"***, che contiene i file java MyKafkaConsumer.java e MyKafkaProducer.java
2) ***"queries"***, che a sua volta contiene 3 packages, uno per ogni query
3) ***"utils"***, contenente classi di supporto per l'implementazione del codice.

3.1. Acquisizione dei dati e Preprocessamento

Lo stream in ingresso viene generato usando un Consumer, che legge i dati dal topic *source* sul quale il Producer ha precedentemente scritto i dati. Una volta letti, i dati vengono trasformati in oggetti di tipo ***Sensor***, una classe creata appositamente contenente le informazioni necessarie al processamento. La classe Sensor rappresenta il sensore analizzato e contiene diversi attributi, come `sensor_id`, `location`, `timestamp`, `temperature`.

Durante la fase di preprocessamento sono stati scartati quei sensori **non aventi riportata la temperatura**, in quanto tale campo era fondamentale per rispondere alle 3 query, mentre ci si è accorti che tutti i sensori **non avevano riportati i valori di altitudine e pressione a livello del mare**, ossia, rispettivamente, i campi `altitude` e `pressure_sealevel`, ma in questo caso si è deciso di porre pari a null tali campi e non scartare i sensori, in quanto tali valori non sono necessari ai fini dell'analisi. Per quei sensori aventi **valori anomali di latitudine e longitudine** si è deciso di porre pari a null tali campi e di non scartare il sensore, poichè per le 3 query tali campi non erano necessari e scartandoli si sarebbero perse informazioni. Un esempio di sensore con un valore anomalo di latitudine è il sensore 69934, avente come latitudine il valore 5.093.518.312. Inoltre, osservando l'output della Query2 ci si è accorti che le temperature medie riportate avevano valori estremamente alti, che superavano i 220° o erano valori negativi molto bassi, che davano a pensare ad ipotetici outliers. Si è scoperto al link https://lastminuteengineers.com/bmp180-arduino-tutorial/ che i sensori BMP180 misurano **valori di temperatura compresi tra -40°C to 85°C**, quindi sono stati filtrati tutti quei sensori aventi un valore di temperatura compreso in tale range, scartando gli altri.

3.1.1. Finestre temporali. Per aggregare i dati su base temporale, ogni query presenta un operatore di **window**, in particolare per tutte e 3 le query è stata utilizzata una **tumbling window**. Le tumbling windows sono finestre di una dimensione specificata, le quali hanno una dimensione fissa e non si sovrappongono. Ad esempio, se si specifica una tumbling window con una dimensione di 5 minuti, verrà valutata la finestra corrente e verrà avviata una nuova finestra ogni cinque minuti, come illustrato nella figura in seguito:



Poichè le tumbling windows in Flink sono allineate con l'epoca 00:00:00 1 January 1970, è stato necessario modificare il parametro offset per allineare le finestre alla prima tupla del dataset, avente come timestamp 2022-05-01T00:00:03, altrimenti la prima finestra iniziava il 2022-04-28T02:00:00. In particolare, si è deciso di far iniziare la prima finestra a 2022-05-01T00:00:00.

3.1.2. Query1. Poichè tale query richiede di calcolare il risultato anche su una finestra temporale dall'inizio del dataset, per poter utilizzare una finestra di 31 giorni (in quanto il dataset è relativo a tutto il mese di Maggio), si è deciso di generare, una volta inviati tutti i dati nel dataset, uno stream "dummy" relativo al 1 Giugno, in modo da poter così determinare una fine per la finestra considerata e poter processare i dati.

Per la query1 sono stati inizialmente filtrati, tramite una ***filter***, tutti quei sensori aventi `sensor_id<10000`, e successivamente tramite una ***keyBy*** è stato partizionato il DataStream per chiave pari all'id del sensore, generando così un ***KeyedStream*** . Successivamente si evoca l'operatore ***window*** sul KeyedStream, in particolare utilizzando finestre temporali di 1 ora, 1 settimana e 31 giorni (durata del dataset), e successivamente si applica l'operatore ***aggregate***, che permette di aggiornare le statistiche della finestra ogni volta che una tupla le viene assegnata. In particolare, l'operatore aggregate implementato consente di contare il numero di misurazioni e il valore medio di temperatura per ogni sensore nella finestra. La funzione di aggregate

utilizzata è ***Q1Aggregate***, che implementa la classe ***AggregateFunction<IN, ACC, OUT>***, dove:

- IN: Il tipo dei valori che vengono aggregati (valori di input),
- ACC - Il tipo di accumulatore (stato aggregato intermedio),
- OUT - Il tipo di risultato aggregato.

In particolare, tale classe prende in ingresso (IN) un oggetto di tipo ***Sensor***, come accumulatore (ACC) un oggetto di tipo ***Q1Accumulator***, e in output (OUT) restituisce un oggetto ***Q1Result***, avente come campi l'id del sensore, la temperatura media, il conteggio e il timestamp della finestra associata.
Successivamente, per poter riportare nei risultati la data corretta di inizio della finestra temporale è si è utilizzata la ProcessWindowFunction ***Q1ProcessWindowFunction*** in modo da recuperare dalla finestra il timestamp relativo all'inizio di quest'ultima. Q1ProcessWindowFunction riceve come input l'oggetto ***Q1Result*** precedentemente creato, gli setta il timestamp della finestra e lo restituisce come output. Tramite una map i risultati poi vengono convertiti in stringhe e infine, tramite un ***sink***, un FlinkKafkaProducer invia l'output del flusso al topic "results" creato appositamente su Kafka.

Il DAG relativo alla Query1 è il seguente:

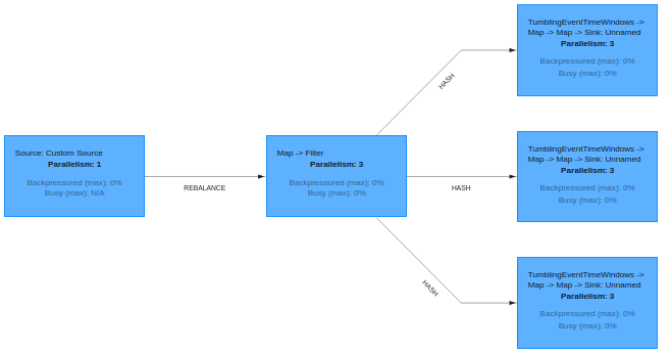


Figure 4. DAG Query1.

3.1.3. Query2. Per quanto riguarda la query2, inizialmente tramite una ***keyBy*** si è partizionato il DataStream per `location`. Dopo aver applicato l'operatore ***window***, su ogni finestra viene applicata la funzione di ***aggregate*** *Q2Aggregate*, che permette calcolare per ogni finestra e per ogni sensore avente lo stesso location id il valore medio della temperatura. Q2Aggregate restituisce un oggetto di tipo ***Q2Result***, il quale ha come attributi `location`, `avg_temperature`, `count` e `timestamp` (il quale verrà assegnato successivamente tramite una ProcessWindowFunction). Poichè la query richiede di calcolare una classifica in tempo realte delle 5 località aventi temperatura media più alta e delle 5 località aventi temperature media più bassa, si è successivamente applicata una ***windowAll*** per generare tale classifica, in modo da unire i Q2Results precedentemente calcolati che appartengono alla stessa finestra, e successivamente è stata applicata la ***ProcessAllWindowFunction*** *Q2ProcessWindowFunction* per poter calcolare la classifica tramite una ***TreeMap*** e recuperare il valore del timestamp relativo alla specifica finestra. L'output di Q2ProcessWindowFunction è una stringa contenente i dati precedentemente calcolati, la quale viene poi passata al ***sink*** per inviare, tramite un ***FlinkKafkaProducer***, l'output del flusso al topic "*results*" creato appositamente su Kafka. Il DAG relativo alla Query2 è il seguente:

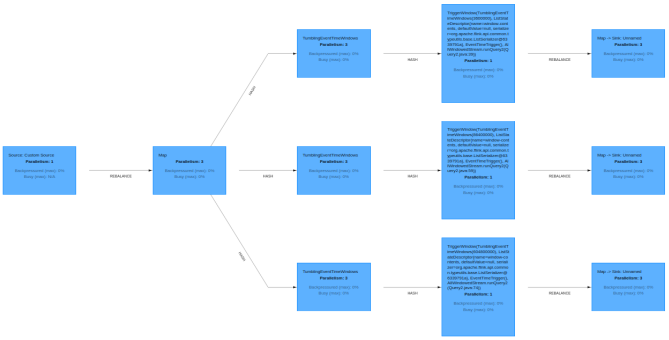


Figure 5. DAG Query2.

3.1.4. Query3. Per rispondere a tale query, per limitare l'area geografica di interesse alle coordinate di latitudine e longitudine (38°, 2°) e (58°, 30°) inizialmente si è costruita, tramite il metodo ***createGrid()*** in *Query3.java*, una griglia di dimensione 4x4 con celle rettangolari, identificando ogni cella da sinistra a destra e dall'alto verso il basso con il nome `cell_X`, dove X è un identificativo da 0 a 15. La griglia risultante è la seguente:

In particolare, tale griglia è stata implementata tramite un *ArrayList<Cell>*, dove Cell è una classe creata appositamente che identifica la singola cella della griglia e avente come attributi:

- *lat1*: latitudine superiore,
- *lat2*: latitudine inferiore,
- *lon1*: longitudine sinistra,
- *lon2*: longitudine destra,
- *idCell*: id della cella

come schematizzato nella seguente figura: Una volta creata la griglia, si assegna ad ogni sensore, tramite il metodo ***setCell()***, la cella a appartiene, e successivamente tramite una ***filter*** si filtrano i sensori che appartengono alla griglia, ossia quelli appartenenti a una tra le 15 celle. Tramite una ***keyBy*** si partiziona il DataStream per cell_id, e successivamente si applica l'operatore ***window***. Su ogni finestra ora viene applicata la funzione di aggregate ***Q3Aggregate***, che permette di mantenere aggiornati, per ogni finestra e per ogni cella, la temperatura media e la temperatura mediana.

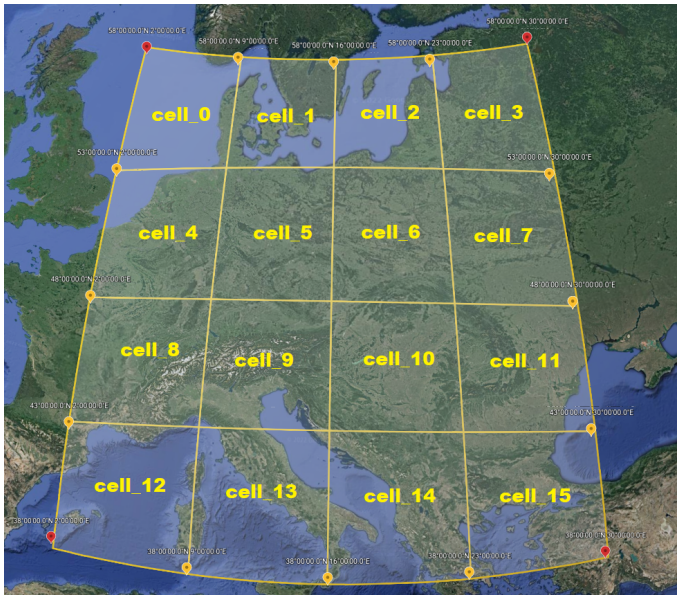


Figure 6. Griglia Query3

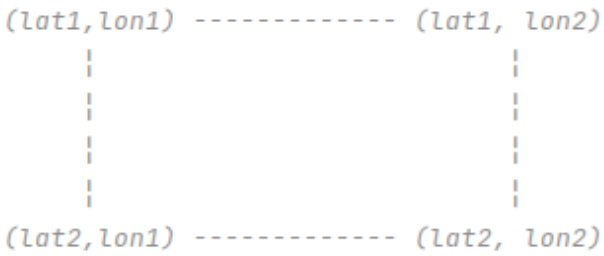


Figure 7. Singola cella

Per il calcolo della mediana in tempo reale, per evitare di ordinare tutti i dati relativi alle temperature e accumulare tutti i valori, sono stati utilizzati un min-heap e un max-heap, strutture dati viste come alberi binari tali che:

- **Max-heap**: un nodo (figlio) non può avere un valore maggiore di quello del suo genitore. Quindi, in un max-heap, il nodo radice ha sempre il valore più grande ,
- **Min-heap**: un nodo (genitore) non può avere un valore maggiore di quello dei suoi figli. Pertanto, in un min-heap, il nodo radice ha sempre il valore più piccolo.

In Java un heap è rappresentato da una *PriorityQueue*. Per calcolare la media, si utilizza un min-heap contenente la metà più grande degli elementi, con l'elemento minimo alla radice, e un max-heap che contiene la metà più piccola degli elementi, con l'elemento massimo alla radice. Quando arriva una nuova temperatura, la si confronta con la radice del min-heap: se tale temperatura è minore della radice (elemento minimo dell'heap), allora lo si inserisce nel max-heap alla posizione opportuna, altrimenti si inserisce nel max-heap. Se, dopo l'inserimento, la dimensione di un heap differisce da quella dell'altro heap di più di 1, possiamo ribilanciare gli heap spostando la radice da un heap all'altro, mantenendo così una differenza di dimensione al massimo di 1. Con questo approccio, possiamo calcolare la mediana come **media delle radici** di entrambi gli heap, se la dimensione dei due heap è uguale. In caso contrario, **la mediana è la radice dell'heap con più elementi**.
Una volta calcolata la mediana, Q3Aggregate restituisce come output un oggetto ***Q3Result***, avente come attributi `cell`, `avg_temperature`, `median`,`count`. Per poter inserire nella stessa finestra le varie celle aventi lo stesso timestamp è stata applicata una ***WindowAll***, e in seguito la ***ProcessAllWindowFunction*** *Q3ProcessWindowFunction*, che restituisce in output un oggetto ***Q3WindowResult***, avente come attributi il timestamp della finestra e una TreeMap<Cell>. A seguire è stata applicata una ***map***, per formattare correttamente i dati da inviare al ***sink***. Il DAG relativo alla Query3 è il seguente:

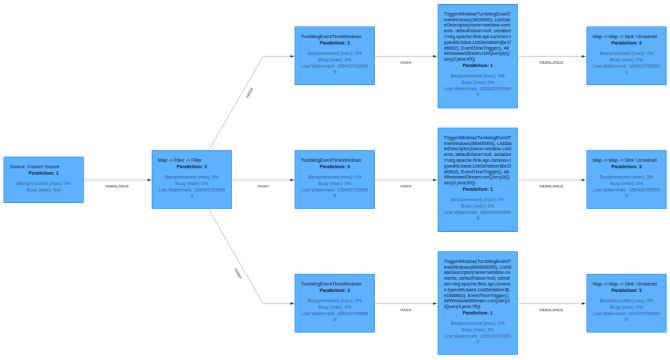


Figure 8. DAG Query3.

4. ANALISI DEI TEMPI

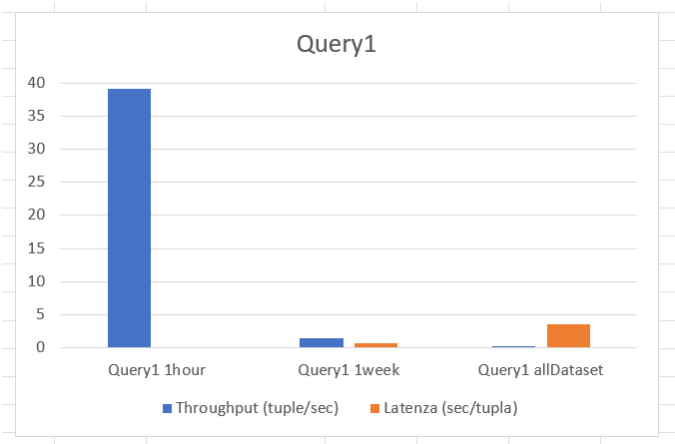
La valutazione delle prestazioni è stata effettuata su sistema operativo Linux Ubuntu 20.04 virtualizzato con a disposizione 8 GB di RAM e 4 core. Per poter eseguire una valutazione sperimentale dei benchmark è stata utilizzata una struttura tenente conto sia del numero di tuple processate che del tempo impiegato; l'accesso a tale struttura 'e effettuato, per evitare inconsistenze nel caso di aggiornamenti simultanei da parte di operatori replicati, mediante l'utilizzo di metodi `synchronized`

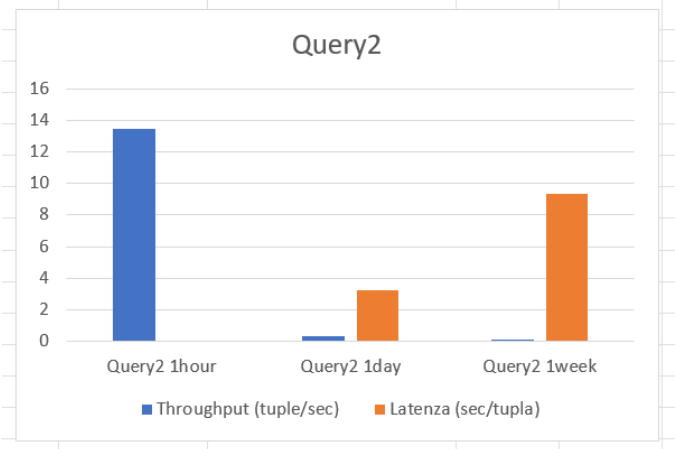
Per calcolare le metriche di throughput e latenza si è utilizzata la classe MyMetricSink, che implementa una MapFunction, la quale viene chiamata tramite una map dopo aver effettuato la computazione. Secondo tale meccanismo, la prima volta che una finestra finisce di processare i dati che contiene e le viene applicata la map, si salva il tempo corrente nella variabile "startTime" tramite `System.currentTimeMillis()`, ossia si salva il tempo del primo evento. Dalla seconda volta, viene incrementato un contatore che tiene conto del numero di tuple processate e calcolato un currentTime pari alla differenza

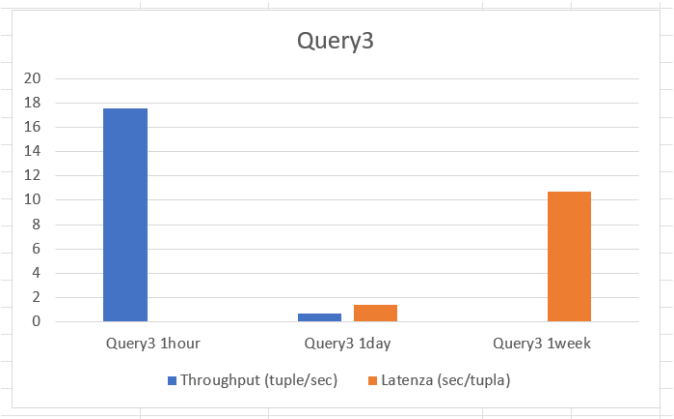
tra il tempo corrente e lo startTime. Il throughput viene calcolato come il rapporto tra il numero di tuple processate e currentTime, quindi calcolato con unità di misura di tuple/sec, mentre la latenza come il rapporto tra currentTime e il numero di tuple processante, quindi con unità di misura di sec/tuple. Di seguito si mostrano i valori di throughput e latenza per le 3 query:

Query	Throughput (tuple/sec)	Latenza (sec/tupla)
Query1 1hour	39.154929577464700	0.0255395683453237
Query1 1week	1.45571861352638	0.686945945945945
Query1 allDataset	0.335954295639689	3.597886354875637
Query2 1hour	13.482941069147966	0.07416779431664412
Query2 1day	0.31080997078386274	3.217455353623
Query2 1week	0.10691756655618517	9.35313467868
Query3 1hour	17.53099097777452	0.0570418410041841
Query3 1day	0.6964596633778293	1.4358333333333335
Query3 1week	0.09353953604390121	10.690666666666667

Figure 9. Throughput e latenza medi delle 3 query al variare dell'event time.







Come si può notare il throughput delle query cresce al diminuire della dimensione della finestra, in linea con quanto ci si aspetti.

References

[1] https://sensor.community/it/

[2] https://archive.sensor.community/csv_per_month/2022-05/2022-05_bmp180.zip