SABD - Progetto 2 2021-2022

Diana Pasquali
Università di Roma Tor Vergata
Roma, Italia
diana.pasquali@alumni.uniroma2.eu

Giacomo Lorenzo Rossi
Università di Roma Tor Vergata
Roma, Italia
giacomolorenzo.rossi@alumni.uniroma2.eu

***Abstract*—L'obiettivo di questo progetto è di eseguire il processamento in streaming di dati raccolti da sensori barometrici.**

***Index Terms*—data stream processing, data ingestion, data visualization, sensors, real time**

I. Introduzione

L'obiettivo di questo progetto è di eseguire il **processamento in streaming** di dati raccolti da sensori barometrici. E' stato utilizzato il dataset di riferimento che contiene i dati relativi al mese di maggio 2022 e tramite un sistema di data ingestion è stato simulato l'arrivo dei dati in tempo reale.

II. Architettura

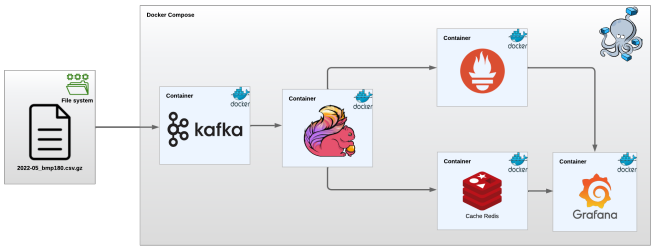

Fig. 1. Deployment e architettura dell'applicazione.

Sono stati utilizzati diversi framework per realizzare questa applicazione. I dati vengono inizialmente recuperati dal file **compresso** `2022-05_bmp180.csv.gz` che viene copiato in un applicazione containerizzata che denominiamo Publisher, viene decompresso e poi riordinato in base al valore del timestamp. Dopodiché il Publisher pubblica ogni record nel topic di **Kafka** `input-records` con un intervallo di tempo tra l'invio di un record e il successivo **proporzionale alla differenza tra i relativi timestamps**, in modo da simulare l'arrivo dei dati in tempo reale a partire dai sensori. Per velocizzare la simulazione, la differenza temporale viene **ridotta di un fattore di velocizzazione** (5.000.000). I record sono consumati direttamente da **Flink**, che usando una sorgente Kafka recupera i dati in arrivo nei Kafka brokers e crea uno stream di dati su cui applica il processamento in tempo reale. Viene poi usato **Prometheus** per memorizzare i dati relativi alle metriche di Flink in termini di **latenza e throughput**, mentre viene usato **Redis** per memorizzare i dati in output dal processamento. Per visualizzare l'output delle differenti query come anche i valori delle metriche raccolte da Flink, viene utlizzato il framework **Grafana**, che si interfaccia con Prometheus e Redis nel serving layer per recuperare le metriche e i risultati delle query **in tempo reale**. Inoltre, per salvare i risultati su file CSV, **Flink** si interfaccia con Kafka in funzione di producer pubblicando i risultati ottenuti su topic diversificati in base al **tipo di query** e al **tipo di finestra** (e.g: query1-Hour) e tali record sono recuperati dal `RecordConsumer` (altra applicazione Java containerizzata) che si occupa di salvare tali record sugli appositi file CSV.

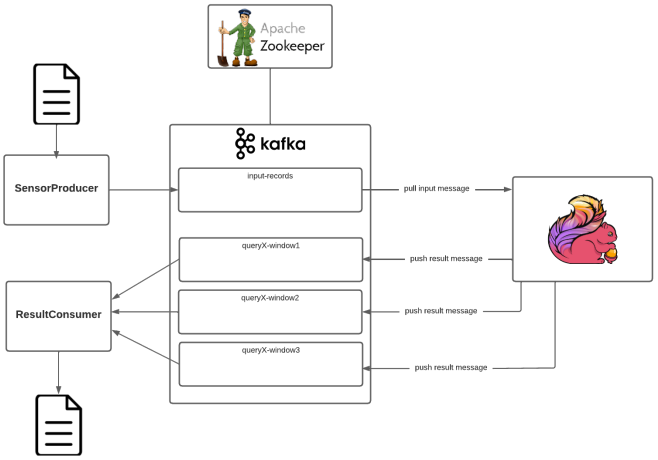

Fig. 2. Architettura di Kafka per la fase di data ingestion e serving layer per la scrittura dei file CSV di risultato.

III. Queries

Ai fini del progetto, sono state implementate le seguenti queries:

- **Query1**: Per i sensori che hanno id ¡ 10000, trovare il numero di **misurazioni totali** e la **temperatura media**. Per calcolare questa query è necessario effettuare il processamento usando delle `Tumbling Windows` basate su Event Time ogni:
  - 1 ora
  - 1 settimana
  - Dall'inizio del dataset
- **Query2**: Trovare la **top-5** delle`location` in tempo reale relativa ai sensori che misurano il **valore medio della temperatura più alto** e la **top-5** delle `location` in tempo reale relativa ai sensori che misurano il **valore**

medio della temperatura più basso. Per calcolare questa query è necessario effettuare il processamento usando delle `Tumbling Windows` basate su Event Time ogni:

  - 1 ora
  - 1 giorno
  - 1 settimana

- **Query3**: Considerare le coordinate di latitudine e di longitudine incluse **nell'area geografica identificata dalle coordinate** pari a `(38°, 2°)` e `(58°, 30°)`. Dividere tale area usando una **griglia 4x4** e identificare ciascuna cella della griglia a partire da quella nell'angolo in alto a sinistra fino a quella nell'angolo in basso a destra, dove l'id della cella va da 0 a 15. Per ogni cella, trovare la **media e la mediana della temperatura** tenendo conto dei valori **emessi dai sensori che sono localizzati all'interno di tale cella**. Per calcolare questa query è necessario effettuare il processamento usando delle `Tumbling Windows` basate su Event Time ogni:
  - 1 ora
  - 1 giorno
  - 1 settimana

A. *Struttura delle query*

Per ognuna delle query considerate, **il filtraggio e la pulizia dei dati avviene direttamente in Flink** a monte nella fase di processamento di ogni singola query, per diversificare la pulizia dei dati in arrivo per ogni query e selezionare le informazioni strettamente necessarie al loro processamento. Inoltre ogni query estende la classe astratta `Query`, che dichiara i metodi seguenti, ma ne lascia l'implementazione alle sue sottoclassi:

- `sourceConfigurationAndFiltering`: metodo che ha il compito di impostare la sorgente della query e di implementare il **pre-processamento** dei record in arrivo da essa.
- `queryConfiguration`: metodo che definisce la topologia del **processamento** della query vera e propria
- `sinkConfiguration`: metodo che ha l'obiettivo di **definire i sink di output** della applicazione, ad esempio Redis. Questo metodo è stato implementato nelle tre query anche per eseguire debugging (con il metodo Flink print()) e il salvataggio su CSV (tramite un sink Kafka, attivo opzionalmente).
- `execute`: questo metodo è implementato direttamente nella classe astratta Query e permette di **avviare l'esecuzione** del processamento.

La struttura delle tre query è quindi molto simile: si inizia prendendo i dati da kafka in un Query#Record (che implementa l'**interfaccia marker** FlinkRecord), si processa il record con la query definita in queryConfiguration() e poi si restituisce un risultato di tipo Query#Result (che implementa l'interfaccia FlinkResult). Questo risultato viene poi salvato su Redis come definito nel metodo `sinkConfiguration()`, grazie alla classe implementata ad-hoc `RedisHashSink<T extends FlinkResult>`, mentre le metriche vengono raccolte dal container di Prometheus automaticamente.

B. *Query 1*

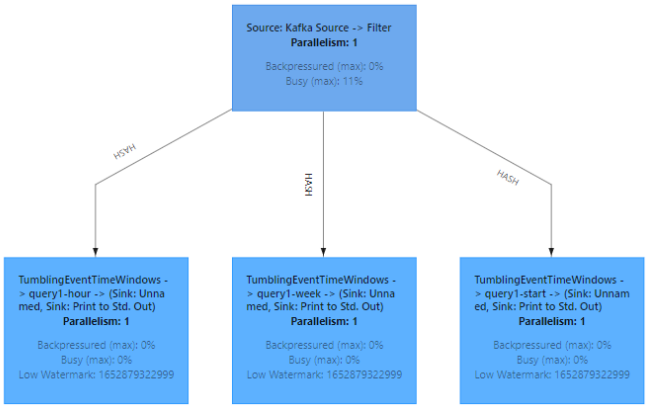

Fig. 3. Topologia del processamento della query 1.

Per prima cosa, vengono recuperati i dati in input a partire dalla sorgente Kafka e **deserializzati** da `QueryRecordDeserializer1`. In tale contesto viene impostata anche la strategia di **Watermarking** da utilizzare (`WatermarkStrategy.<Query1Record>forBoundedOutOfOrderness(Duration.ofSeconds(60))`. In questo modo si riesce a gestire eventuali record fuori ordine con un ritardo massimo di 1 minuto. Inoltre, viene anche assegnato un timestamp ai diversi record, prendendo in considerazione quello fornito dal record stesso. Dopodiché, tramite una funzione `filter` è possibile applicare il pre-processamento, definito nello specifico all'interno della classe `RecordFilter1`. In questo contesto, vengono selezionati i sensori in base al valore dell'id (`sensor_id < 10000`) e sono **considerati esclusivamente i valori di temperatura compresi tra -93.2 e 56.7 °C** che sono rispettivamente i valori della minima temperatura e della massima temperatura registrate di sempre e si escludono tutti i record che hanno dei valori nulli nelle colonne di interesse. Si procede perciò con il processamento effettivo dello stream dei dati ottenuto in output dal `filter`. Si **seleziona la chiave** (il `sensor_id`) tramite la funzione `keyBy` partizionando il flusso di dati in base ad essa. Si imposta la finestra di tipo `TumblingWindowEventTime` con tutti e tre i possibili valori temporali (`Hour`, `Week` e `FromStart`), quindi si calcolano i risultati su più finestre in contemporanea (perché nella `main()` il metodo di processamento `queryConfiguration()` viene chiamato 3 volte, con le rispettive finestre). Per calcolare il valore della media si applica un' `AggregateFunction` (che implementiamo in `AverageAggregator1`) congiunta a una `ProcessWindow` (`Query1ProcessWindowFunction`) per **sommare incrementalmente i valori di temperatura e il conteggio** delle misurazioni e restituire in output **al termine della finestra il valore medio della temperatura e il**

conteggio aggregato in un oggetto di tipo `Query1Result`. Infine vengono **raccolte le metriche di throughput e latenza** tramite una `RichMapFunction` e viene ritornato il flusso di dati in output. Tali dati in output, vengono distribuiti a dei `Sink` che permettono di salvare i dati su Redis in tempo reale, mentre le metriche vengono lette dal framework Prometheus configurato nel file `docker-compose.yml`, per generare delle time-series per la latenza e il throughput.

C. Query 2

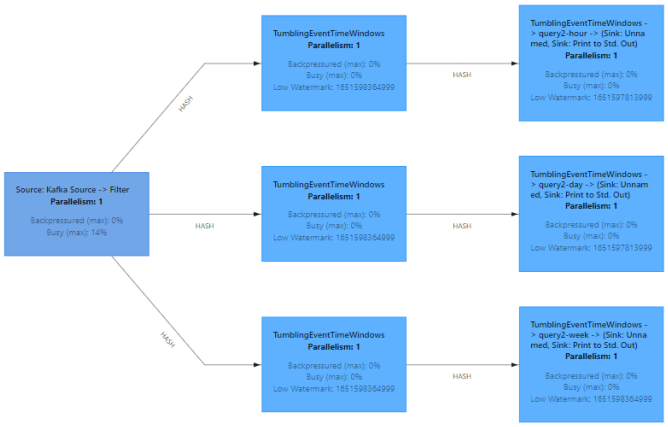

Fig. 4. Topologia del processamento della query 2.

Nella query2 lo stream di dati viene recuperato dalla **sorgente Kafka**, utilizzando come **deserializzatore** la classe `QueryRecordDeserializer2` e viene successivamente impostata la strategia di **Watermarking** in maniera equivalente alla Query 1, **assegnando** tramite il metodo `withTimestampAssigner` **il valore del timestamp presente in ogni record** prodotto dai sensori. Per poter procedere nel processamento, è necessario effettuare una pulizia dei dati, eliminando i record che hanno dei valori fuori misura della temperatura o che hanno valore nullo nei campi che sono di interesse per la query stessa. I valori importanti per questa query sono la **temperatura e l'id della** `location`. In particolare si controlla che la **temperatura sia compresa tra -93.2 °C e 56.7 °C**, come per la query1, e inoltre si controlla che il valore della `location` non sia nullo. Tutto questo viene effettuato tramite una `filter()` che abbiamo implementato nella classe di pre-processamento `RecordFilter2`. Dopodiché il flusso di dati passa al **processamento** vero e proprio, implementato nel nostro metodo `queryConfiguration()`. A questo punto i dati vengono **partizionati in base al valore della** `location`, dal momento che è necessario calcolare il valore medio della temperatura per le `location` che cadono all'interno della specifica finestra temporale. In modo analogo alla query1, le tre finestre temporali (`Hour`, `Day`, `Week`) vengono calcolate nell'ambito dello stesso ambiente di esecuzione Flink in contemporanea, chiamando tre volte `queryConfiguration()` nel `main()` e variando il parametro relativo alla dimensione della finestra. In tutti e tre i casi, si procede al **calcolo delle temperature medie** utilizzando una `ProcessWindowFunction` con **aggregazione incrementale**. Perciò alla API Flink `aggregate()` è necessario passare come valori di input due istanze: una istanza di `AverageAggregate2` che permette di calcolare le medie delle temperature per ogni `location` (che internamente sostituiamo al sensor_id, come richiesto nella traccia) e una `Query2ProcessWindowFunction` che permette di accedere al contesto di esecuzione di Flink e di ottenere un valore **univoco** di timestamp per tutti i risultati della computazione nell'ambito di una singola finestra. Dopodiché usiamo una `windowAll` per raggruppare **tutti i risultati ottenuti in un'unica finestra** con 10 locazioni e temperature, massime e minime. Riassegnando perciò nuovamente il valore della finestra, è possibile trovare il ranking delle temperature tramite un'altra chiamata al metodo Flink `aggregate()`. In particolare, a tale funzione è necessario fornire come valori di input un'istanza di `RankAggregate`, che sfrutta al suo interno una classe `RankAccumulator`, la quale mantiene le due top5 all'interno di due `TreeSet` di massimo 5 elementi, delle strutture dati che **permettono di mantenere ordinato l'insieme anche dopo l'aggiunta di nuovi valori**. Ogni volta che arriva un nuovo valore di temperature, se questa temperatura è maggiore della 5° temperatura massima nella finestra corrente, viene aggiunto nel TreeSet delle temperature massime, mentre se è minore della 5° temperatura minima, viene aggiunto nel TreeSet delle temperature minime. In entrambi i casi, la temperatura che non fa più parte della top-5 corrispondente viene eliminata. Al termine della finestra, viene restituita in output un'istanza di `Query2Result` che contiene i valori dei `sensor_id` relativi alle `location` della top 5 e bottom 5 con le rispettive temperature medie. Infine, segue la raccolta delle metriche per Prometheus e il salvataggio dei dati su Redis in un hash, implementando la nostra classe astratta `RedisHashSink<Query2Result>`.

D. Query 3

1) Definizione della griglia: Per realizzare il processamento di questa query, è necessario dapprima definire la griglia 4x4 specificata nella richiesta. In particolare è necessario **suddividere l'Europa centrale in 16 celle** in un'area compresa tra le coordinate (38°, 2°) e (58°, 30°). Per ottenere questo numero di celle, a partire dalla coordinata dell'angolo in basso a sinistra sono stati aggiunti 7° alla latitudine e 5° alla longitudine (Figura 2) ed in questo modo sono state ottenute le coordinate dei punti che delimitano ciascuna delle 16 celle. Per rappresentare nel codice la griglia e le celle sono state implementate le 4 classi `GeoGrid`, `GeoCell`, `GeoSegment` e `GeoPoint`:

- `GeoGrid` Rappresenta la griglia geografica 4x4 ed è composta da 16 GeoCell. Le celle sono numerate da 0 a 15, partendo dalla cella a SUD-OVEST e crescendo prima orizzontalmente e poi verticalmente, arrivando alla cella 15 a NORD-EST. Al suo interno sono stati implementati due metodi: `getContainingCell` per **ricavare la GeoCell di appartenenza di un punto oppure nulla se**

è esterno alla griglia e `insideGrid` per **determinare se un punto è interno alla griglia o meno**.

- `GeoCell` Rappresenta la cella geografica nella griglia 4x4 ed è **identificata da 4** `GeoPoint` (latitudine e longitudine) e da una **lista di segmenti inclusi** (`GeoSegment`). La **convenzione** da noi attuata prevede che i **punti che appartengono ai segmenti ai bordi sud e ovest della cella sono inclusi di default nella cella**. Invece i punti che si trovano sui bordi nord appartengono alla cella immediatamente superiore a quella considerata, se presente, altrimenti sono di competenza della cella stessa. In maniera analoga, i punti che si trovano sui bordi est, appartengono alla cella immediatamente a est rispetto a quella considerata, anche in questo caso solo se presente, altrimenti anche questi punti sono di competenza della cella stessa. In base a questa convenzione perciò, anche **i punti che si trovano sugli angoli a nord-ovest e sud-est non sono inclusi nella cella considerata**, ma appartengono rispettivamente alla cella **immediatamente superiore e immediatamente ad est rispetto alla cella considerata, sempre se tali celle siano effettivamente presenti**, altrimenti vengono gestiti dalla stessa cella. La classe ha un metodo `containsGeoPoint` che permette di verificare se un punto geografico appartiene alla cella, rispettando la convenzione appena descritta.
- `GeoSegment` rappresenta un segmento delimitato da due `GeoPoints`. La particolarità è che i due punti **estremi possono essere anche esclusi dal segmento** e questo ci è utile per definire i confini delle celle seguendo la convenzione da noi pensata. Include un metodo `containsPoint` che **permette di determinare se un punto appartiene al segmento** o meno.
- `GeoPoint` rappresenta semplicemente una posizione geografica con **latitudine e longitudine**

Tutti i metodi presenti nelle classi `GeoGrid`, `GeoCell` e `GeoSegment` sono stati implementati con un **approccio Test Driven** (con test in JUnit5) per migliorare la confidenza riguardo alla loro correttezza.

2) Implementazione: Dopo aver definito la griglia e le classi di utilità per gestirla, la query è stata implementata in `queryConfiguration` **mappando ogni misurazione di un sensore nella sua cella di appartenenza** all'interno della griglia con la classe `CellMapper`, che sfrutta il metodo `getContaingCell(GeoPoint)` per mappare da `Query3Record` a `Query3Cell`, che contiene una GeoCell, la temperatura, il timestamp e l'id del sensore presente in Query3Record. E' possibile però che il sensore sia al di fuori della griglia: in tal caso `getContainingCell()` restituisce un `Optional.empty()`, quindi bisogna **filtrare via tutti i punti che non appartengono a nessuna cella** (usiamo semplicemente un `filter(p -> p.getCell != null)`). Dopodiché **raggruppiamo per id della cella** con `keyBy` e assegnamo la dimensione finestra (presa dall'input di queryConfiguration(), al solito chiamato per le tre finestre `Hour`, `Day` e `Week`). Successivamente utilizziamo una AggregateFunction

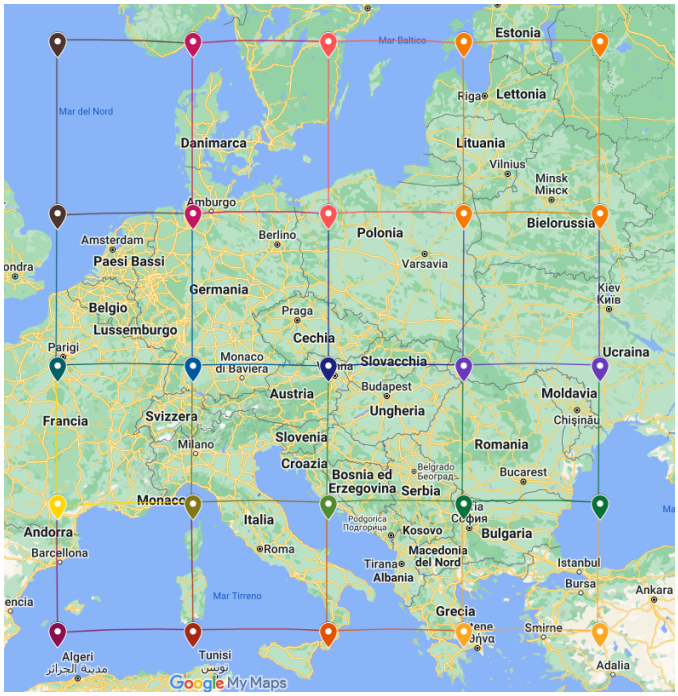

Fig. 5. Griglia utilizzata per la Query3. I colori dei bordi rappresentano l'insieme dei punti presi in considerazione nei casi limite per ciascuna cella.

implementata da `AvgMedianAggregate3` per calcolare la media e la varianza dei Query3Cell relativi a ogni cella (che ricordiamo è la chiave) e salva il risultato in un bean `CellAvgMedianTemperature` che contiene timestamp, GeoCell e temperature media e mediana **per una singola cella**. Il calcolo della media è analogo alle query precedenti, mentre la **mediana è stata calcolata in modo approssimato e on-line usando l'algoritmo** P^2 nella classe `P2MedianEstimator`, modificata per essere compatibile con le AggregateFunction di Flink. In particolare, a partire dall'implementazione C# di Andrey Akinshin (vedi riferimenti), abbiamo aggiunto un metodo merge() per unire due P2MedianEstimator in uno solo mantenendo comunque una **buona approssimazione della mediana senza mantenere in memoria tutte le misurazioni**. Dopo un ultimo `windowAll` per ottenere un'unica finestra raggruppando tutte e 16 le celle, chiamiamo la `process` implementata con `FinalAllProcessWindowFunction` per mettere tutti i dati delle 16 `CellAvgMedianTemperature` in una singola classe `Query3Result` e ottenere l'output richiesto: in particolare se non ci sono misurazioni per una cella, la `FinalAllProcessWindowFunction` crea delle istanze vuote di `CellAvgMedianTemperature` con valori NaN per media e mediana. Infine usiamo una `MetricRichMapFunction` per raccogliere latenza e throughput e impostiamo dei Sink su Redis per le tre finestre (come nelle query precedenti).

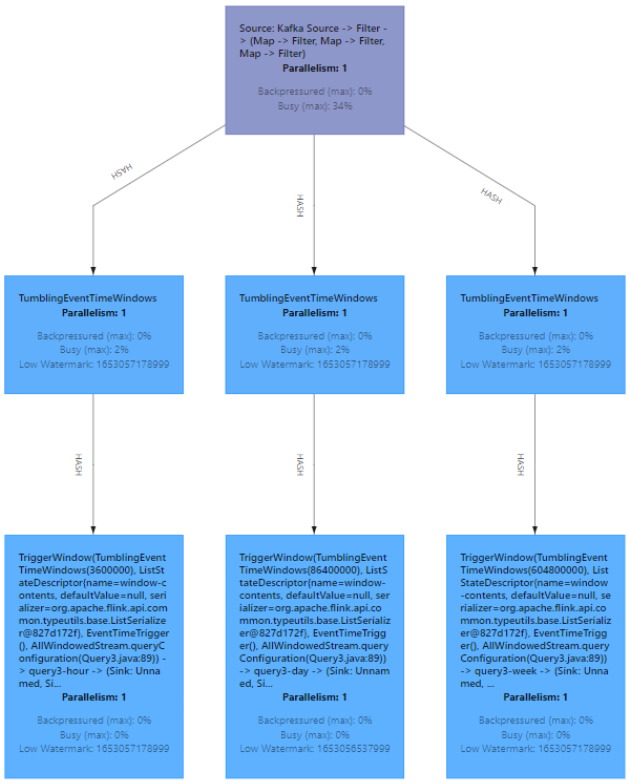

Fig. 6. Topologia del processamento della query 3.

E. Query 1 con Kafka Streams

Si è scelto di implementare la Query 1 utilizzando anche il framework **Kafka Streams** come lavoro opzionale. La scelta è stata effettuata tenendo in considerazione che per la fase di data ingestion è stato utilizzato Kafka, e dunque per evitare di aggiungere un ulteriore container è stato preferito a Spark Streaming. Anche in questo caso, come anche nell'implementazione delle query con Flink, vengono recuperati i dati direttamente dal topic Kafka `input-records` e vengono impostati i parametri per la deserializzazione nelle proprietà di Kafka utilizzando un `Serde` implementato ad-hoc per ricevere valori provenienti dal CSV di tipo `SensorDataModel`. La fase di **filtraggio dati** avviene utilizzando direttamente la funzione `filter()` di Kafka Streams e con gli stessi criteri applicati anche nella Query 1 con Flink. Il **processamento** della query viene definito all'interno del metodo `queryKafka()` che prende in input anche i parametri relativi alle finestre da impostare. Il processamento è ripreso dalla query 1 Flink, applicando in questo caso le API di Kafka Streams. Utilizzando una `groupBy` si **divide lo stream di dati in input in base alla chiave** su cui calcolare i valori medi della temperature (nel caso di questa query la chiave è `sensor_id`). Dopodiché si **imposta il valore della finestra sulla base del parametro** passato in input al metodo di definizione della topologia. Una volta definita la durata della finestra, utilizzando la funzione `TimeWindows.ofSizeWithNoGrace(Duration)` si crea una **Tumbling Window** e si procede aggregando valore per valore le tuple in arrivo, istanziando dei nuovi oggetti di tipo `CountAndSum()`. Ognuno di questi ha la funzione di aggregatore per i valori sommati delle temperature e delle occorrenze per tutte le tuple appartenenti alla finestra precedentemente specificata e per tutte le tuple che hanno un determinato id del sensore. Dopo una suppress per l'eliminazione dei risultati parziali, viene eseguita una `mapValues` che trasforma gli aggregati in tipi `AvgResult` calcolando contestualmente anche il valore della media. Inoltre viene trasformato il tipo `AvgResult` in stringa, sfruttando il metodo toStringCSV del bean `AvgResult` per salvare nella maniera corretta i risultati sui file CSV.

IV. Visualizzazione

Per realizzare la visualizzazione dei risultati delle 3 queries è stata utilizzata una **dashboard interattiva di Grafana**. In particolare, tramite i diversi `Sink` i dati in output dalle query sono memorizzati all'interno di Redis, che funge da Serving Layer dell'applicazione.

A. Redis

Il salvataggio dei dati su Redis, avviene mediante una `RichSinkFunction` **implementata ad hoc**, chiamata `RedisHashSink`, in quanto la classe ufficiale di Flink non permetteva di assegnare agli Hash Redis più di un campo alla volta. Sfruttando la **libreria di Redis per Java, Jedis**, la classe **astratta** `RedisHashSink` permette di memorizzare i dati in arrivo al sink all'interno di HashSet di Redis, affinché possano essere recuperati successivamente dal framework di visualizzazione. Ogni query utilizza una classe separata `RedisHashSink#` che estende la classe `RedisHashSink` per salvare in modo differenziato i dati di output su Redis.

B. Grafana

Per ogni query, viene visualizzata un'informazione differente:

- **Query1** Tramite le apposite variabili Grafana, **è possibile selezionare il tipo di finestra** (`window type`) e **l'id del sensore** da ispezionare (`sensor id`) e dunque viene visualizzato in tempo reale il **valore medio della temperatura e il numero di misurazioni raccolte** nella finestra temporale selezionata.
- **Query2** Tramite l'apposita variabile Grafana, **è possibile selezionare il tipo di finestra** (`window type`) e vengono visualizzati **con grafici a barre aggiornati in tempo reale la top 5** delle `location` con le **temperature medie maggiori** e la **top 5** delle `location` con le **temperature medie minori**. Vengono anche visualizzati i `sensor_id` relativi ai luoghi le cui temperature fanno parte delle due top5.
- **Query3** Tramite l'apposita variabile Grafana, **è possibile selezionare il tipo di finestra** (`window type`) e dunque vengono **visualizzati in tempo reale con due grafici a barre** i valori delle **temperature medie e mediane per ciascuna delle 16 celle** della griglia

di osservazione, in funzione dell'intervallo temporale specificato.

Fig. 7. Dashboard Grafana con i risultati delle tre query aggiornati in tempo reale

C. Prometheus

Come anticipato, oltre ai risultati delle query, Grafana è **stato utilizzato anche per visualizzare l'andamento delle metriche di throughput e latenza**, sfruttando la sua compatibilità con Prometheus per visualizzarle in **due grafici a serie temporali**. Anche in questo caso sono state sfruttate le variabili Grafana (e le regex) per poter visualizzare solo un tipo di finestra o tutte quelle disponibili per la query in esecuzione.

V. Valutazione delle Performance

La piattaforma di riferimento usata per eseguire le query ha queste caratteristiche:

- Windows 10 Home
- Docker con WSL2
- Intel Core i5 10a generazione, 4 core, 8 thread
- 8 GB RAM (2 GB RAM per docker)

I valori sono stati presi dai grafici Grafana in Prometheus. Notiamo come **per le finestre di tipo Hour la latenza è molto più bassa rispetto a tutte le altre finestre (TABLE II)**, mentre **il throughput è molto più elevato (TABLE I)**. C'era

Fig. 8. Grafico Grafana con Throughput e Latenza della query1 nelle finestre Hour, Week e FromStart

Fig. 9. Grafico Grafana con Throughput e Latenza della query2 nelle finestre Hour, Day e Week

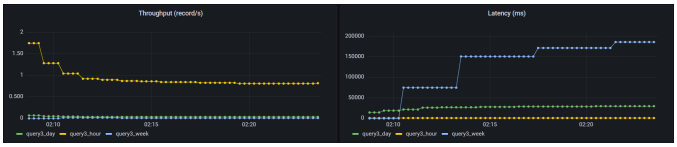

Fig. 10. Grafico Grafana con Throughput e Latenza della query3 nelle finestre Hour, Day e Week

da aspettarsi questo risultato in quanto per le finestre Hour la frequenza di uscita dei risultati è molto più elevata rispetto a Day, Week o addirittura FromStart. Notiamo che per la query 1 con finestre FromStart sia latenza che throughput sono 0 finché la simulazione non termina: comunque in tal caso la latenza è molto alta e il throughput medio complessivo molto basso. Inoltre, confrontando le prestazioni medie della Query 1 in Flink e Query 1 in Kafka Streams, è possibile notare come le seconde siano migliori anche se di poco. Questo risultato potrebbe essere dovuto al fatto che il processamento in Kafka Streams avviene sullo stesso container da cui vengono recuperati i dati in input.

TABLE I
THROUGHPUT (TUPLE/S)

Tipo query	**Tipo finestra**		
	hour	***week***	***start***
Query 1	8,7024	0,223934783	0,0148
Query 1 KafkaStreams	12,912	0,087	0,0199
	hour	***day***	***week***
Query 2	0,97392	0,041348	0,007857826
Query 3	0,986103448	0,0413	0,007953704

TABLE II
LATENZA (MS)

Tipo query	**Tipo finestra**		
	hour	***week***	***start***
Query 1	186,04	14009	67640
Query 1 KafkaStreams	77,443	11486,90	50056,65
	hour	***day***	***week***
Query 2	1034	24512	133932,1304
Query 3	1029,62069	24603,89655	133611,5556

References

- Dashboard Grafana (risultati query): http://localhost:8000/d/2J8ln097k/sabd-project-2?orgId=1&var-redis=Redis-Cache&var-sensor=8762&var-window=Hour
- Dashboard Grafana (prestazioni query): http://localhost:8000/d/UDdpyzz7z/prometheus-2-0-stats?orgId=1&refresh=5s
- Documentazione Docker: https://docs.docker.com/
- Documentazione Kafka: https://kafka.apache.org/documentation/
- Documentazione Flink: https://nightlies.apache.org/flink/flink-docs-master/
- P^2 Quantile Estimator: https://aakinshin.net/posts/p2-quantile-estimator-adjusting-order/