SABD - Progetto 2

Esecuzione di Query Flink su dataset “Sensor Community BMP180”

Danilo Dell’Orco
Facoltà di Ingegneria
Università di Roma Tor Vergata
Roma, Lazio, Italia
danilodellorcolp@gmail.com

Jacopo Fabi
Facoltà di Ingegneria
Università di Roma Tor Vergata
Roma, Lazio, Italia
jacopo.fabi1997@gmail.com

Michele Salvatori
Facoltà di Ingegneria
Università di Roma Tor Vergata
Roma, Lazio, Italia
michelesalvv@gmail.com

ABSTRACT

L’obiettivo del progetto è quello di analizzare tramite Data Stream Processing i dati forniti da Sensor Community riguardo le misurazioni di temperatura effettuate dai sensori, utilizzando il dataset *“2022-05 BMP180”* fornito in formato *csv*. Tramite Kafka si leggono progressivamente i dati dal dataset e si inseriscono tramite stream verso Flink. Da Flink vengono eseguite le varie query sulle diverse finestre temporali, ed i risultati vengono salvati in csv sul taskmanager e poi in locale.

1. INTRODUZIONE

Ai fini del progetto è necessario eseguire 3 operazioni di analisi sul dataset messo a disposizione.

1. Calcolare la media della temperatura misurata da tutti i sensori con id inferiore a 10000.
2. Calcolare la media della temperatura misurata in ogni zona, e identificare le cinque zone con temperature più alte e le cinque zone con temperature più basse.
3. Suddividere i sensori in una griglia $4x4$ con vertici $(38°, 2°)$ e $(58°, 30°)$. Per ognuna delle 16 celle calcolare la media e la mediana della temperatura.

Questo documento ha lo scopo di descrivere nel dettaglio l’architettura utilizzata nel progetto ed i punti principali della sua implementazione, motivando le varia scelte progettuali effettuate.

Il documento è strutturato come segue: la sezione 2 descrive i diversi componenti/frameworks del sistema e relativi dettagli implementativi, la sezione 3 descrive la logica di gestione degli errori, la sezione 4 spiega l’implementazione delle tre query, la sezione 5 descrive la fase di testing e analisi delle prestazioni, la sezione 6 conclude il documento con le considerazioni finali sul sistema proposto.

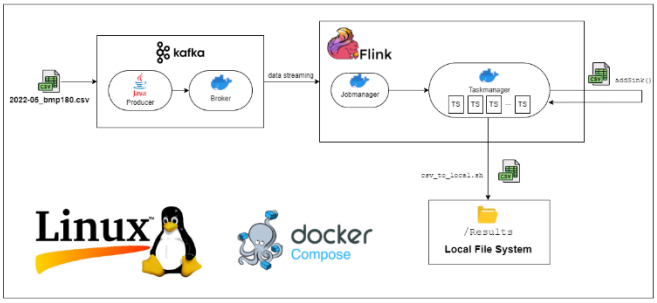


Figura 1 - Architettura di Sistema

2. ARCHITETTURA E IMPLEMENTAZIONE

Per l’analisi del dataset si utilizzano diversi framework per obiettivi differenti. L’esecuzione delle query avviene sul sistema locale, sfruttando la virtualizzazione a livello di sistema operativo offerta da **docker**. Ognuno dei framework dello stack utilizzato viene quindi hostato su uno o più container docker, e tutti questi container sono interconnessi tra loro, facendo parte della stessa rete virtuale. L’orchestrazione dei container avviene tramite `docker-compose`. Si analizzano quindi nel dettaglio i componenti architetturali del sistema considerato *(figura1)*.

- Apache Kafka (*confluentinc/cp-kafka* [1])
- Kafdrop (*obsidiandynamics/*kafdrop [2])
- Apache Flink (*flink:latest* [3])
- Apache Zookeeper (*confluentinc/cp-zookeeper:latest* [4])

2.1. Kafka

Apache Kafka è una piattaforma per il data streaming distribuita che permette di pubblicare, sottoscrivere, archiviare ed elaborare flussi di record in tempo reale. È progettata per gestire flussi di dati provenienti da più sorgenti distribuendoli a più consumatori.

Nel nostro progetto, Kafka permette di leggere il dataset in formato csv, e di simulare le rilevazioni dei sensori per l’analisi dello stream in tempo reale. Lo stream viene utilizzato quindi come sorgente da Flink per l’esecuzione delle query.

Per Kafka si utilizza un *singolo broker* su container docker, mentre il container *Kafdrop* fornisce l'accesso ad una WebUI per monitorare lo stato dei topic.

2.1.1. Implementazione

Il file `Producer.java` realizza il producer del Sistema. Questo legge dati dal dataset csv, effettua un parsing in un messaggio di tipo stringa, e posta tali messaggi sul topic `flink-events` o `kafka-events` a seconda del framework (Flink o KafkaStreams). Per simulare l'istante di rilevazione (`timestamp` nel csv) come l'istante di inserimento del messaggio sul topic si calcola la differenza temporale tra un record e quello successivo, e si mette il thread in sleep per quel lasso temporale. Si utilizza uno speeding factor di 360000 in modo da velocizzare la simulazione; in particolare; due misurazioni a distanza di un'ora verranno postate sul topic a distanza di 10 millisecondi.

Inoltre, al singolo Record prodotto sul topic Kafka viene assegnato il timestamp in cui è avvenuta la misurazione, così come accadrebbe in uno scenario reale. In questo modo, non sarà necessario utilizzare uno specifico `TimestampAssigner` in Kafka e Flink utilizzerà come event-time dal record del topic, bensì verrà direttamente utilizzato il timestamp associato al messaggio.

2.1.2. Zookeeper

Apache ZooKeeper è un framework utilizzato da diversi progetti open-source per garantire una maggiore affidabilità delle risorse del cluster e ulteriori servizi di configurazione del cluster stesso, svolgendo servizi centralizzati per il mantenimento dei metadati delle varie applicazioni in esecuzione nella sua rete.

Le principali funzionalità che ZooKeeper svolge nel cluster proposto sono:

- *Elezione del Kafka Broker leader*, assicurando che ulteriori broker prendano il ruolo di leader nel caso in cui il corrente leader subisse dei fallimenti
- *Topic Configuration*: mantiene la lista di tutti i topic presenti incluse le informazioni relative al numero di partizioni per ogni topic, locazione delle repliche e ulteriori dettagli.

2.1.3. Kafdrop

Kafdrop è una web UI che permette all'utente la visualizzazione dei vari Kafka Topic presenti nel cluster e le loro principali informazioni come il numero di broker, il numero di partizioni, i consumer attivi su un determinato topic. Infine, kafdrop permette di visualizzare i messaggi scritti sul topic.

È stata utilizzata per agevolare lo sviluppo dell'applicazione e contemporaneamente per offrire una rapida via di accesso verso risultati computati dal framework *KafkaStreams* semplicemente utilizzando un browser web.

2.2. Flink

Apache Flink è un sistema di processamento distribuito per computazione stateful su streams di dati bounded e unbounded, quindi progettato sia per batch che stream processing. Flink è stato progettato per essere un sistema in grado di offrire un alto throughput, ed una bassa latenza.

Si utilizza un'immagine docker per il *jobmanager*, ed un'altra immagine docker per il *taskmanager*. Si è deciso di utilizzare un unico taskmanager, ognuno con massimo 12 task slot assegnabili, in quanto si è visto sperimentalmente che il jobmanager tende a schedulare i task su un unico taskmanager; considerando il grado di parallelismo richiesto dall'applicazione non è stato quindi necessario l'utilizzo di più taskmanager. [5]

Inoltre, negli scenari reali tipicamente si utilizza un taskmanager per ogni server fisico, quindi un unico taskmanager nel nostro ambiente locale.[6]

2.2.1. Deserializer

Contemporaneamente alla scrittura delle misurazioni lette dal dataset dal `Producer.java`, l'applicazione di Data Stream Processing legge i nuovi messaggi scritti sul topic deserializzandoli in oggetti `Event`, permettendo l'accesso facilitato ai vari valori della misurazioni tramite i metodi *getter/setter*.

A tale scopo è stato definito un `EventDeserializer` che implementa l'interfaccia `DeserializationSchema` offerta dalle `API` di Flink. Tramite l'`override` del metodo `deserialize()` vengono trasformati i byte letti dal topic in singole istanze di `Event`, definendo tutti gli attributi presenti nel dataset per ogni misurazione. Inoltre, al momento della *deserializzazione*, vengono validati i dati, come descritto nel dettaglio nella errori.

2.2.2. Watermarks

Come strategia di watermark si utilizza `forMonotonousTimestamps`. In questo modo si prende come *event time* il timestamp relativo al messaggio kafka associato, ovvero l'istante di misurazione riportato nella linea del file `csv`.

2.2.3. Salvataggio dei Risultati: CSV Sink

I risultati di ogni query vengono direttamente salvati in locale in formato `csv` durante l'esecuzione dell'applicazione. Ciò avviene utilizzando uno `StreamingFileSink` che permette di scrivere i file partizionati sui vari filesystems supportati da Flink, andando ad effettuare l'encoding dei singoli oggetti ricevuti in righe distinte del file finale.

Nell'applicazione in esame ogni query realizzata in Flink restituisce come computazione finale un oggetto che implementa l'interfaccia `OutputQuery` ed il suo metodo `toCSV()` attraverso il quale si produce una stringa compatibile con l'output atteso per la specifica query.

In questo modo, ogni `StreamingFileSink` esegue l'encoding di un oggetto `OutputQuery` tramite l'encoder custom `CSVEncoder.java`, che si limiterà a scrivere sullo stream in output la stringa del risultato restituita dal metodo `toCSV()`.

2.2.4. Windowing

Per l'esecuzione delle query sono state definite finestre di 1 ora, 1 giorno, 1 settimana ed 1 mese.

Le finestre orarie e giornaliere vengono definite semplicemente utilizzando `TumblingEventTimeWindows.of(DURATION)`.

Invece per le finestra di una settimana e un mese è necessario considerare che flink suddivide il tempo in finestre della durata specificata a partire dall'epoca Unix (01-01-1970). [7]

Ciò significa ad esempio che una finestra di 31 giorni ricadrebbe nel periodo `(14-04 : 15-05)` e non in modo automatico nel range desiderato `(01-05 : 01-06)`. Per ovviare a tale problema, nel definire le finestre *settimanali* e *mensili* è stato applicato un offset rispettivamente di 3 e 17 giorni per allineare l'inizio della Tumbling Window con l'inizio del dataset (01-05-2022).

```
TumblingEventTimeWindows.of(Time.days(7),Time.days(3))
TumblingEventTimeWindows.of(Time.days(31),Time.days(17))
```

2.3. Kafka Streams

L'applicazione di DSP implementata tramite il framework KafkaStreams effettua la lettura dei messaggi dal topic Kafka, scritti dal `Producer.java` per simulare una reale sorgente di dati. I dati vengono deserializzati in istanze di oggetti `Event` utilizzando la classe `EventSerde` che implementa le interfacce `Serializer` e `Deserializer` fornite da Kafka. Queste interfacce seguono la stessa logica descritta nel precedente paragrafo 2.2.1.

2.3.1. Windowing

Anche Kafka, cosi come Flink, utilizza delle finestre temporali implementate dalla classe `Windows<TimeWindow>` che sono allineate all'epoca Unix, ovvero al 01-01-1970. Per allineare l'inizio delle finestre al 01-05 sono state definite due finestre custom `MonthlyWindow` e `WeeklyWindow` che, tramite il metodo `windowsFor()` della classe `Windows` di Kafka, a partire dal timestamp di ogni singolo evento che viene processato restituiscono una finestra della durata esatta desiderata, calcolata a partire dall'event-time e non dall'epoca Unix.

3. GESTIONE ERRORI

Nonostante il dataset sia ordinato, in uno scenario reale i dati potrebbero arrivare nel sistema malformati, fuori ordine, oppure con valori non appartenente al loro dominio reale.

Per questo nel sistema sono state progettate diverse eccezioni in modo tale da rendere il processamento dei dati robusto, e più orientate ad una esecuzione *fire & forget*.

3.1. Valori fuori Ordine

I valori fuori ordine vanno gestiti sia nel producer Kafka che nell'analisi con Flink.

Nel caso di Kafka tali valori non vengono scartati, ma devono essere gestiti in quanto una differenza negativa tra due record porta un errore nel `thread.sleep()`. A tale scopo è stata definite l'eccezione `SimulationTimeException()` che semplicemente setta a 0 il tempo per cui il thread andrà in sleep.

Nel caso di Flink invece le tuple fuori ordine possono essere gestite con un ritardo massimo che varia in base alla specifica finestra temporale considerata, sfruttando il parametro `allowedLateness` della finestra.

3.2. Temperature fuori Range

I sensori `BMP180` considerati nel dataset possono misurare temperature che variano tra $-40°$ e $+85°$ [8]. Ciò vuol dire che le temperature fuori da tale range sono da considerare errate, e vanno ignorate ai fini dell'analisi. Per questo durante la fase di deserializzazione, si verifica se la temperatura supera i limiti considerati: se i limiti sono rispettati, allora viene generato un nuovo `Event`, altrimenti viene catturata una `TemperatureOutOfBoundException` e non viene istanziato un evento relativo a quel record. Flink, infatti, gestisce nativamente eventuali record nulli, semplicemente ignorandoli [9].

3.3. Latitudine e Longitudine fuori Range

La latitudine può avere un valore compreso tra $-90°$ e $+90°$, mentre la longitudine tra $-180°$ e $180°$. Eventuali valori fuori da questo range sono da considerarsi errati dunque e vanno ignorati. Nel `try/catch` del `deserialize`, se viene catturata una `CoordinatesOutOfBoundsException` il record viene ignorato e non si istanzia il relativo evento.

4. QUERY

Vediamo le scelte effettuate durante la progettazione delle varie query.

4.1. Query 1

Per i sensori con sensor_id <10000, trovare il numero di misurazioni e il valore medio della temperatura. Utilizzando una tumbling window, calcolare la query per ogni ora , ogni settimana e dall'inizio del dataset.

4.1.1. Flink

Si utilizza una `filter()` per mantenere i soli eventi con id del sensore inferiore a 10000. Successivamente con `keyBy()` si specifica il `sensor_id` come chiave dello stream. Tramite `window()` si specificano le diverse `TumblingEventTimeWindow` da considerare. Tramite `aggregate()` si applica sulla finestra l'`AggregateFunction AvgQ1`, che calcola la media della temperatura per ogni chiave. Il risultato fornito dal suo metodo `getResult()` è un oggetto `OutQ1`, che mantiene i soli campi necessari per l'output, ovvero `timestamp`, `sensor_id`, `count` e `mean_temperature`. I vari oggetti OutQ1 vengono infine scritti su file `CSV` tramite l'apposito `StreamingFileSink`.

4.1.2. Kafka Streams

I messaggi, una volta deserializzati in `Event`, subiscono un'operazione di filtraggio tramite la quale vengono esclusi dal processamento le misurazioni effettuate da sensori aventi `sensor_id<10000`.

Successivamente, tramite l'operatore `map()`, gli eventi vengono mappati in oggetti ausiliari `ValQ1` che mantengono l'ID del sensore che ha effettuato la misurazione, il timestamp in cui essa è avvenuta, il valore rilevato e il numero di occorrenze che viene inizializzato ad 1, utilizzato successivamente per il conteggio delle occorrenze.

All'operazione di mapping segue l'operatore `selectKey()` che andrà a selezionare l'ID del sensore come chiave di ogni evento che compone lo stream.

Le operazioni appena descritte compongono un `KStream<Long, ValQ1>` al partire dal quale verranno effettuate parallelamente le computazioni sulle tre differenti finestre di eventi richieste.

Dal `KStream` gli eventi vengono raggruppati per chiave, ovvero per `sensor_id`, tramite l'operatore `groupByKey()`, al quale segue la definizione della finestra basata sull'event-time, tramite l'operatore `windowedBy()`.

Successivamente, l'operatore `reduce()` andrà ad effettuare il conteggio delle occorrenze e la somma dei valori della temperatura rilevata da ogni singolo sensore nella fascia oraria desiderata, andando a sommare tra loro istanze diverse di `ValQ1` che sono state raggruppate per `sensor_id`. Infine, un'ultima operazione di `map()` effettuerà il calcolo della temperatura media, portandoci alla definizione di una `KTable<Long, ValQ1>` che rappresenta il risultato finale.

4.2. Query 2

Trovare la classifica in tempo reale delle prime 5 località con la temperatura media più alta e la classifica delle prime 5 località con la temperatura media più bassa media. Utilizzando una tumbling window, calcolare la query per ogni ora , ogni giorno ed ogni settimana.

4.2.1. Flink

A partire dalla sorgente kafka, si utilizza una `keyBy()` per specificare il `location_id` come chiave dello stream. Tramite `window()` si specificano le diverse `TumblingEventTimeWindow` da considerare, e tramite `aggregate()` si applica sulla finestra l'`AggregateFunction AvgQ2`, che calcola la media della temperatura per ogni chiave, fornendo oggetti `ValQ2` in output.

A seguito dell'aggregazione si ha un oggetto `SingleOutputStreamOperator<ValQ2>` contenente tutte le medie della temperatura per ogni `location_id`.

Successivamente tramite `process()` si applica allo stream la `ProcessAllWindowFunction LocationRanking`, che per ogni finestra calcola le cinque zone con temperatura media più alta e le cinque con temperatura media più bassa. A tale scopo nel metodo `process()`, la funzione user-defined `getLocationsRanking`, che scorre la lista di `iterable`, genera le liste con le temperature più alte e più basse, e le ritorna all'interno di un oggetto `OutQ2`.

I vari oggetti `OutQ2` vengono infine scritti su file `CSV` tramite l'apposito `StreamingFileSink`. [8.2]

4.2.2. Kafka Streams

Come per la precedente query i dati vengono letti dal topic Kafka utilizzando il deserializzatore `EventSerde`. In seguito verrà selezionata come chiave dello stream l'identificatore della `location` dell'evento, tramite l'operatore `selectKey()`, ottenendo dunque un `KStream<Long, Event>`.

Il `KStream` appena descritto viene raggruppato per chiave, ovvero vengono raggruppate tutte le misurazioni effettuate in una specifica location, tramite la trasformazione `groupByKey()`, alla quale segue poi la definizione della finestra temporale, tramite `windowedBy()`, sulla quale effettuare le computazioni.

Successivamente, tramite la trasformazione `aggregate()`, vengono calcolate le occorrenze e la media delle misurazioni che si verificano in quella determinata location, utilizzando una classe ausilaria `ValQ2` che funge da accumulatore dei risultati calcolati per ogni diversa location.

Ottenute le temperature medie registrate nelle diverse location per ogni finestra temporale, esse verranno raggruppate in base alla finestra cui appartengono. Ciò è possibile poiché, nella precedente aggregazione, vengono utilizzate le funzioni `getWeekSlot/getDailySlot/getHourSlot` per impostare un `timestamp` univoco associato all'accumulatore `ValQ2` di ogni `location`. Ad esempio, considerando una finestra temporale settimanale, un accumulatore `ValQ2` per la location `54321` nella seconda settimana del mese, avrà `timestamp=2022-05-08`. In questo modo, applicando la trasformazione `groupBy()` su tale `timestamp`, verranno processati dalla successiva trasformazione tutti gli accumulatori per quella *settimana*.

Tale processamento viene effettuato da un'ultima operazione di `aggregate()` che andrà a restituire un oggetto `LocationAggregator` per ogni finestra temporale. Questo mantiene due `TreeMap<Double,Long>` che rappresentano la lista delle *5 location* con temperatura media più alta e la lista delle *5 location* con temperatura media più bassa. È stato scelto di utilizzare le `TreeMap` poichè, utilizzando come `Key` delle `Entry` il valore medio di temperatura e come Value il `LocationID`, tramite il metodo `put()` viene mantenuto l'ordinamento ascendente delle chiavi, ovvero le liste saranno mantenute ordinate in base alla temperatura media.

Per ogni accumulatore `ValQ2` che entrerà nella fase di aggregazione, verrà invocato il metodo `updateRank()` del `LocationAggregator` che aggiornerà le liste delle migliori 5 location sfruttando il metodo `put()` delle `TreeMap`. Infine, come nella precedente query, i risultati vengono scritti su un *Kafka Topic* tramite l'operatore `to()`.

Query 3

Considerare le coordinate di latitudine e longitudine all'interno dell'area geografica identificata dalle coordinate di latitudine e longitudine (38°, 2°) e (58°, 30°).

Suddividere quest'area usando una griglia 4x4 ed identificare ogni cella della griglia dall'angolo in alto a sinistra a quello in basso a destra usando il nome "cell_X", dove X è l'id della cella da 0 a 15.

Per ogni cella, trovare la temperatura media e la mediana, tenendo conto dei valori emessi dai sensori che si trovano all'interno di quella cella.

Utilizzando una tumbling window, calcolare la query per ogni ora, ogni giorno ed ogni settimana.

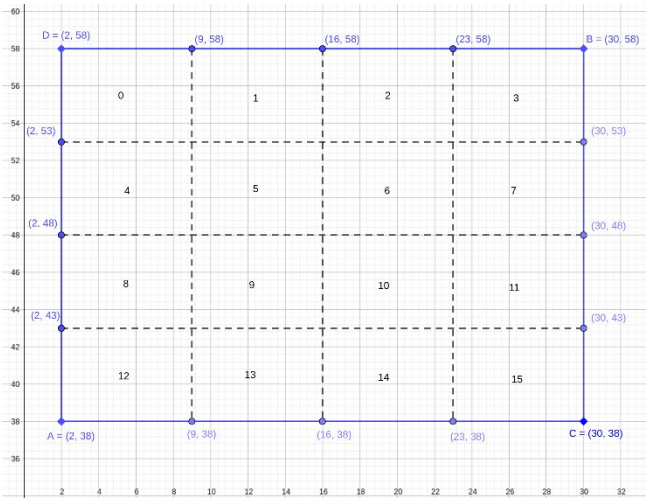


Figura 2 – Griglia 4x4

Definizione della griglia e assegnazione dei sensori

Si istanzia innanzitutto un oggetto `Grid`, specificando la latitudine e la longitudine dei vertici top-left e bottom-right. Nel costruttore di questo oggetto viene invocato il metodo `split()`, che suddivide la griglia in una griglia 4x4, generando quindi 16 oggetti `Cell`. Ogni cella è identificata dai due vertici e dal suo id, che varia da 0 a 15 *(figura 2)*.

Successivamente si utilizza una `filter()` per mantenere i soli sensori appartenenti alla griglia. A tale scopo si utilizza nella *lambda function* una funzione booleana `isSensorInGrid()` che verifica se le coordinate del sensore rientrano nei limiti definiti dalla griglia.

In seguito tramite una `map()` si associa ogni evento ad un oggetto `ValQ3`, che contiene i campi relativi al timestamp, alla temperatura (media e mediana) e all'identificativo della cella cui appartiene il sensore. Per assegnare il sensore alla sua cella si utilizza il metodo `getCellFromEvent()`, che scorre la lista di celle della griglia, e verifica il sensore in quale di questa ricade.

Infine, si esegue una `keyBy()` per specificare il `cell_id` di ogni sensore come chiave dello stream. Alla fine di questo processo quindi ogni sensore è identificato dalla specifica cella nel range `0,15` a cui appartiene*(figura 3)*.

Calcolo media e mediana

A partire dal `KeyedStreams<ValQ3> keyed` definito in precedenza, si applica la `TumblingEventTimeWindow` tramite `window()`, e successivamente si ottiene la media della temperatura per ogni cella tramite `aggregate()` con `AggregateFunction AvgQ3`. Nel metodo `getResult()` viene ritornato un oggetto `ValQ3` in cui la temperatura media è settata sulla media calcolata, mentre la mediana è impostata su `null`.

La mediana viene calcolata in maniera esatta per ogni finestra temporale. Per fare ciò si applica la `WindowProcessFunction`

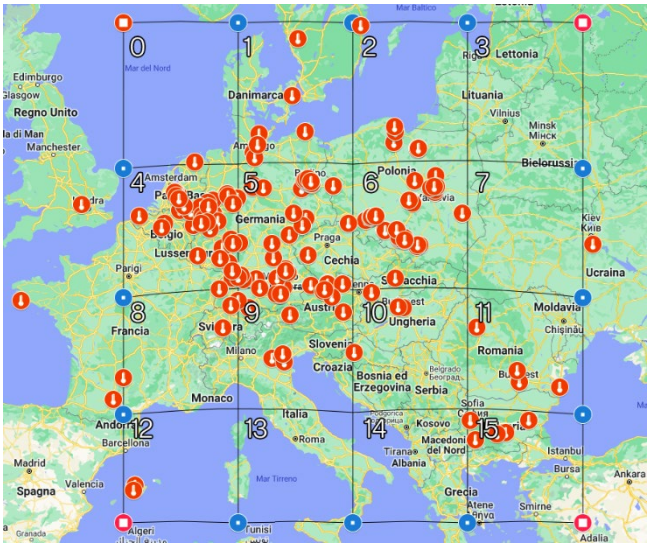


Figura 3 – Assegnazione dei sensori nella griglia (Google My Maps [10])

`Median` a partire dall'oggetto `keyed` definito in partenza. Nel metodo `process()` si ordina la lista di rilevazioni nella cella in base alla temperatura, e successivamente si ottiene la mediana come l'elemento centrale della lista (per lista di dimensione parti) o come la media dei due elementi centrali (per lista di dimensione dispari).

Risultato Finale

Dopo aver calcolato media e mediana, si hanno due `SingleOutputStreamOperator<ValQ3>` distinti, chiamati rispettivamente `mean` e `median`. Per fornire i risultati finali su un'unica linea del csv si esegue quindi il `join` dei due streams, specificando il `cell_id` come campo su cui unire i due streams.

Successivamente si utilizza `apply()` per indicare una `JoinFunction`, che permette di mantenere a seguito del join i soli campi `cell_id`, `mean_temp`, `median_temp` e `timestamp`.

A questo punto si ha un `DataStream<ValQ3>`, in cui ogni record presenta media e mediana della temperatura nella specifica cella. I dati per ogni cella nella specifica finestra devono essere forniti su un'unica linea, per cui si applica una ulteriore process function `CellStatistics`, che semplicemente prende tutti i ValQ3 della finestra (quindi media e mediana di ogni cella), e li restituisce in un unico oggetto `OutQ3`. In questo modo si ottieneun `SingleOutputStreamOperator<OutQ3>` result in cui ogni record presenta le statistiche di ogni cella nella specifica fascia oraria.

Infine, quando viene eseguito il sink per la scrittura del csv viene invocato il metodo `toCSV()` di `OutQ3`, e quindi il csv viene scritto nel formato desiderato. [8.3]

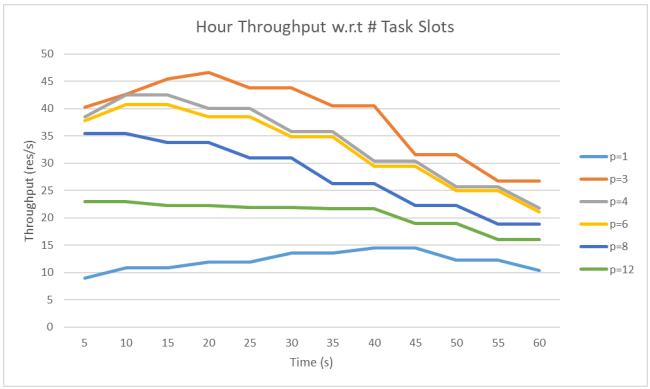


Figura 4 – Throughput al variare del numero di task slots

5. TESTING E ANALISI DELLE PERFORMANCE

Per analizzare le prestazioni del sistema è stato prima individuato il grado ottimale di parallelismo, e successivamente sono state testati i tempi di risposta per ognuna delle query.

La macchina utilizzata per il testing ha le seguenti specifiche:

```
AMD Ryzen 5 5500U with Radeon Graphics, @6x2.10 GHz, RAM 8GB DDR4 @1593.9 MHz, Linux 5.15.23-76051523-generic amd64 (Pop!_OS by System76).
```

5.1. Monitoraggio delle Prestazioni

Per monitorare le prestazioni del sistema, si tengono traccia delle metriche di *latenza* e *throughput*. Entrambe vengono monitorate tramite un apposito script python, che sfrutta le *REST API* di flink per richiedere periodicamente le misurazioni.

5.1.1. Latenza

La latenza misurata è intesa end-to-end, e si sfruttano i `LatencyMarker` di flink con intervallo di tracking di 1 s tramite `setLatencyTrackingInterval(1000)`.

I valori della latenza vengono misurati dall'operatore Kafka Source fino al sink per la scrittura del csv, quindi misurando il tempo che intercorre tra l'immissione del dato nel sistema, e la produzione del risultato.

Per monitorare i valori della latenza nel tempo si avvia lo script python di monitoraggio, che periodicamente ottiene i valori della metrica interrogando il seguente URL:

```
localhost:8081/jobs/<jobID>/metrics?get=latency.source_id.<KafkaID>.operator_id<SinkID>.operator_subtask_index.0.latency_<statistic>
```

In questo modo si possono ottenere *media*, *varianza* e *percentili* della latenza. I valori misurati vengono scritti in un apposito file csv, man mano che si registrano nuovi valori.

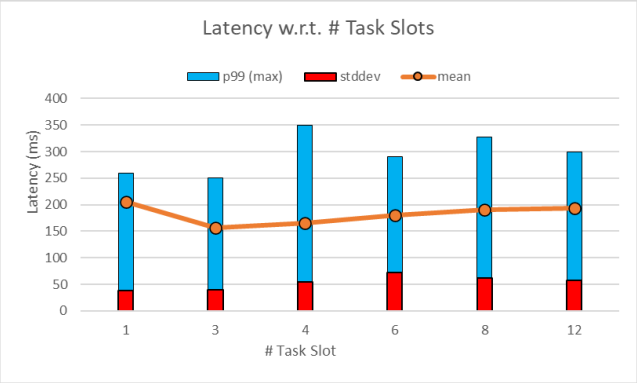


Figura 5 – Throughput al variare del numero di task slots

In Kafka Streams invece, non è stata effettuata una misurazione della *latenza end-to-end* in quanto le metriche presenti effettuano tale misurazione andando a comparare il *timestamp di injection dei record* con il *timestamp* del sistema al momento del loro *processamento*. Tuttavia, ai record inseriti dal producer sul topic è stato associato l'effettivo timestamp in cui il sensore ha effettuato la misurazione; pertanto, non effettuando il processamento in tempo reale ma solo una simulazione di DSP a distanza di mesi, la comparazione dei timestamp prima citati con i timestamp di processamento avrebbe dato luogo ad una misurazione erronea della latenza.

5.1.2. Throughput

Il throughput viene misurato in termini di *risultati prodotti al secondo*. Si definisce quindi un oggetto `Metric`, in cui viene invocato il metodo `markEvent()` ogni volta che viene prodotto un nuovo risultato. Pertanto, è stata definita la metrica user-defined `Throughput` su tutti gli operatori precedenti al nodo sink per la scrittura del csv.

Per monitorare i valori del throughput nel tempo si avvia lo script python di monitoraggio, che periodicamente ottiene i valori della metrica interrogando il seguente URL:

```
localhost:8081/jobs/<jobID>/vertices/<operatorID>/
subtasks/metrics?get=<operatorName>.Throughput
```

I valori del throughput progressivamente misurati vengono scritti in un apposito file csv.

La misurazione del throughput in KafkaStreams viene effettuata utilizzando le ProcessAPI, delle API di basso livello che permettono di definire processori custom e connetterli ad altri processori all'interno della topologia che definisce l'applicazione DSP. Al momento dell'istanziazione del processore custom `MetricProcessor` verrà definita una nuova metrica custom attraverso la definizione di un `Sensor`, aggiungendola alle metriche già presenti nel `ProcessorContext` di Kafka, che rappresenta il numero di record che il processore vede passare. Ogni volta che un risultato viene computato, passa nel processore appena definito, il quale provvederà ad aggiornare la metrica appena

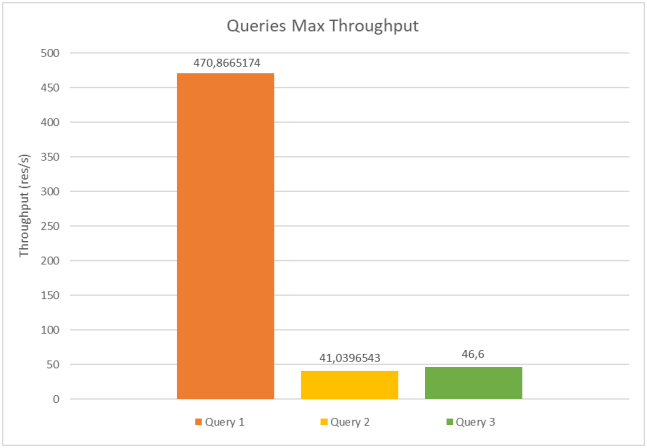


Figura 6 – Throughput delle query Flink

descritta. Parallelamente, un thread `MetricCalculator` calcolerà ogni 5 secondi il throughput della query in esame. Ciò viene fatto accedendo alla metrica definita in precedenza e alla metrica `start-time-ms` definita da Kafka per ottenere il timestamp di inizio processamento dell'applicazione. In questo modo il throughput potrà essere calcolato andando a dividere il numero di record visti dal processore in quel momento, per il tempo trascorso.

5.2. Numero di Task Slot

Per individuare il numero ideale di task slot è stato utilizzato un approccio sperimentale, andando a testare throughput e latenza al variare del grado di parallelismo utilizzato. In particolare, è stato valutato il comportamento della query 3, essendo quella più complessa, e considerando media, varianza e 99° percentile.

Analizzando i risultati forniti dal grafico (*figura 4*) è possibile vedere come con un solo task slot si ha un throughput di massimo 15 risultati prodotti al secondo, mentre il throughput massimo si raggiunge con 3 task slots. Continuando invece ad aumentare il grado di parallelismo, il throughput inizia progressivamente a decrescere, in quanto si va a saturare la macchina locale su cui flink è in esecuzione.

Anche analizzando la latenza *(figura 5)* possiamo vedere come si ottengano i risultati migliori utilizzano 3 task slots. Infatti, si misura una latenza media di 156ms, una varianza di 40ms ed il 99° percentile (che corrisponde al massimo) è di 251ms.

5.3. Testing delle Query

Per testare le prestazioni delle query, vengono eseguite diverse ripetizioni di ogni query, resettando l'ambiente di esecuzione al termine di ogni ripetizione. In questo modo sono state valutate le prestazioni a parità di condizioni in ognuno dei test, ripristinando ogni volta la sorgente di data stream.

Per effettuare un confronto prestazionale tra le query, è stato ricavato il tempo medio di risposta a partire dal throughput *(figura6)*. Questo perché il numero di risultati prodotti al secondo, dipende anche dal numero di risultati che ogni query dovrà produrre.

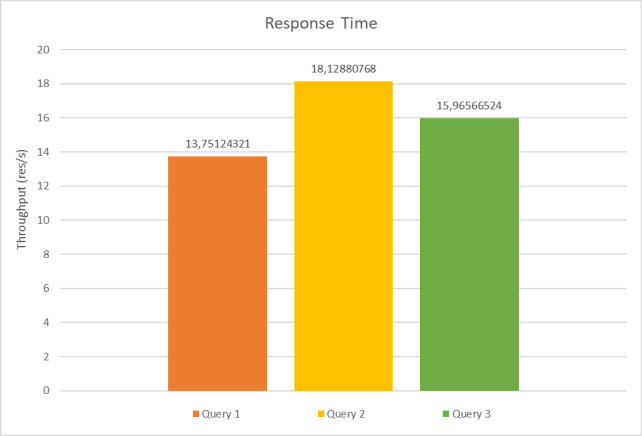


Figura 7 – Confronto tempi di risposta query Flink

Pertanto, si ricava una stime del tempo di risposta:

$$Ts = \frac{\#\ risultati\ attesi\ [res]}{throughput\ [res/s]}\ [s]$$

Come possiamo vedere dal grafico (*figura 7*), complessivamente la query 2 è quella che ha il tempo di risposta maggiore. Questo è un comportamento atteso in quanto è l'unica query che opera sull'intero dataset, e quindi le computazioni sulle singole finestre impiegheranno più tempo. La query 3 al contrario ha un tempo di risposta leggermente inferiore alla query 2. Questo perché, nonostante richieda molte più computazioni per la gestione della griglia ed il calcolo della mediana, applicando la filter si riduce notevolmente il numero di sensori da considerare escludendo quelli fuori griglia.

5.4. Confronto Flink – Kafka Streams

Confrontando i Throughput ottenuti nelle prime due query Flink con quelli delle query Kafka Streams *(figura 8)* è possibile vedere come Flink risulti 5 volte più prestante nel caso della prima query, e addirittura 12 volte più prestante nel caso della seconda query. Questo è ragionevole in quanto Flink è progettato per essere un framework a bassa latenza e ad alto throughput, mentre Kafka Streams è più orientato all'affidabilità e alla fault-tolerance. Inoltre, Kafka Streams risulta poco efficiente nel caso di operazioni con stato, e questo giustifica le scarse prestazioni della seconda query, in cui si utilizzano due operatori di aggregazione.

6. CONCLUSIONI

6.1. Punti di Forza

Le query in Flink risultano molto efficienti in termini di throughput e latenza. Infatti, considerando il dataset a disposizione e lo speeding factor introdotto, i risultati delle finestre vengono computati in parallelo e restituiti in un tempo che varia dai 13 ai 35 secondi a seconda della query e della finestra temporale.

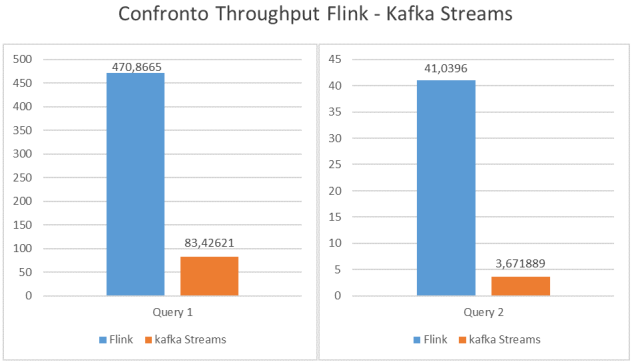


Figura 8 – Confronto Flink – Kafka Streams

Inoltre, l'applicazione risulta robusta a possibili errori nei dati immessi dallo stream, quindi orientata all'utilizzo in uno scenario reale.

Gli script di monitoraggio realizzati fanno utilizzo delle Rest API, e permettono di monitorare l'applicazione anche in uno scenario reale su stream unbounded, per individuare colli di bottiglia ed errori nel sistema.

6.2. Limitazioni

Le principali limitazioni sono presenti nell'utilizzo di *KafkaStreams*, principalmente in relazione alle prestazioni delle query che presentano un throughput notevolmente inferiore rispetto a quelle Flink.

Inoltre, per quanto riguarda le API di monitoraggio, la metrica di misurazione della latenza end-to-end offerta da Kafka non è utilizzabile nel nostro scenario DSP simulato.

Infine, KafkaStreams è vincolato all'utilizzo di Kafka come sorgente dati, al contrario di Flink che permette di integrare come sorgente anche altri servizi.

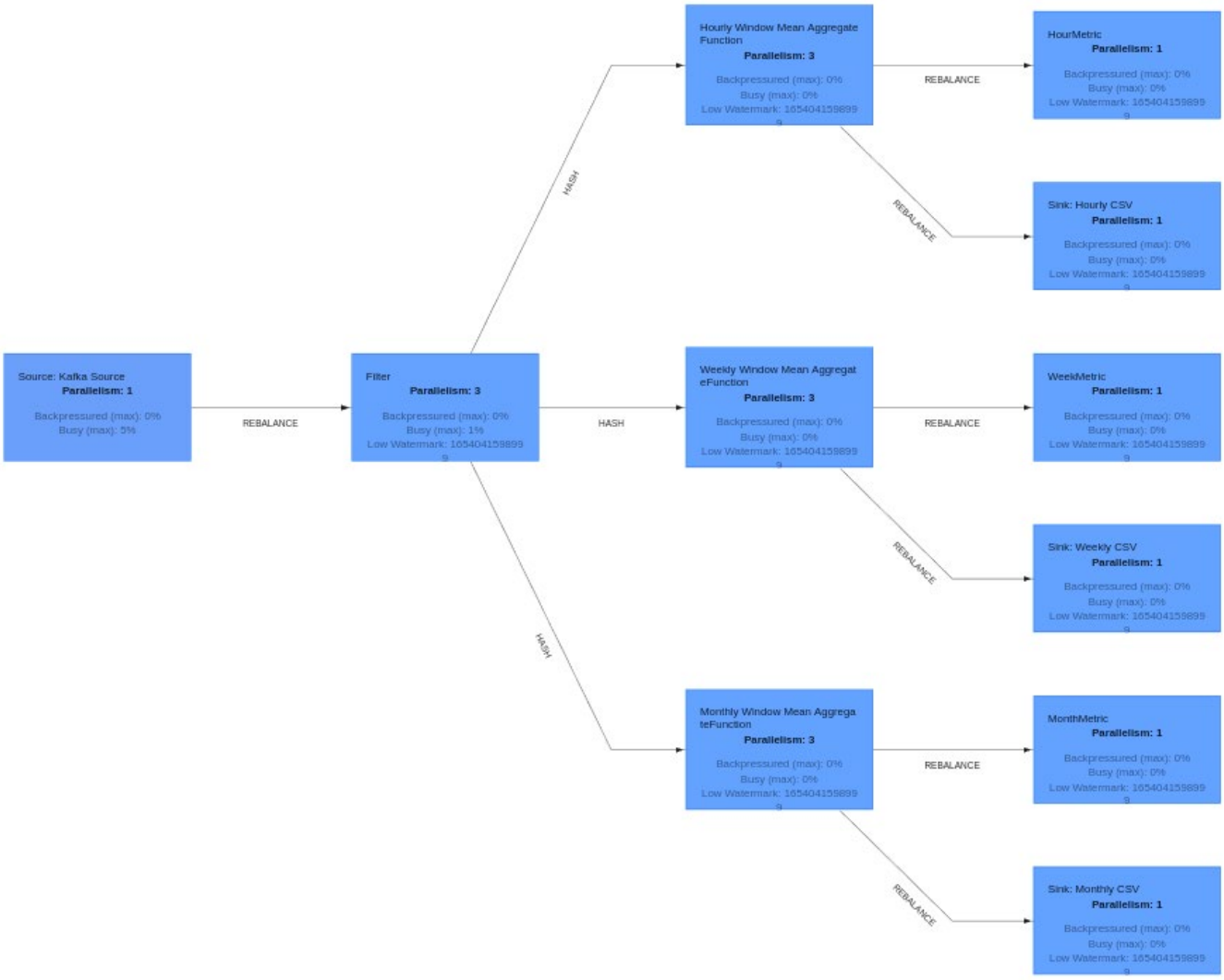
Avendo utilizzato una soluzione basata su containerizzazione non sono stati sfruttati a pieno i vantaggi del processamento parallelo, avendo come collo di bottiglia le risorse computazionali della macchina utilizzata.

7. RIFERIMENTI

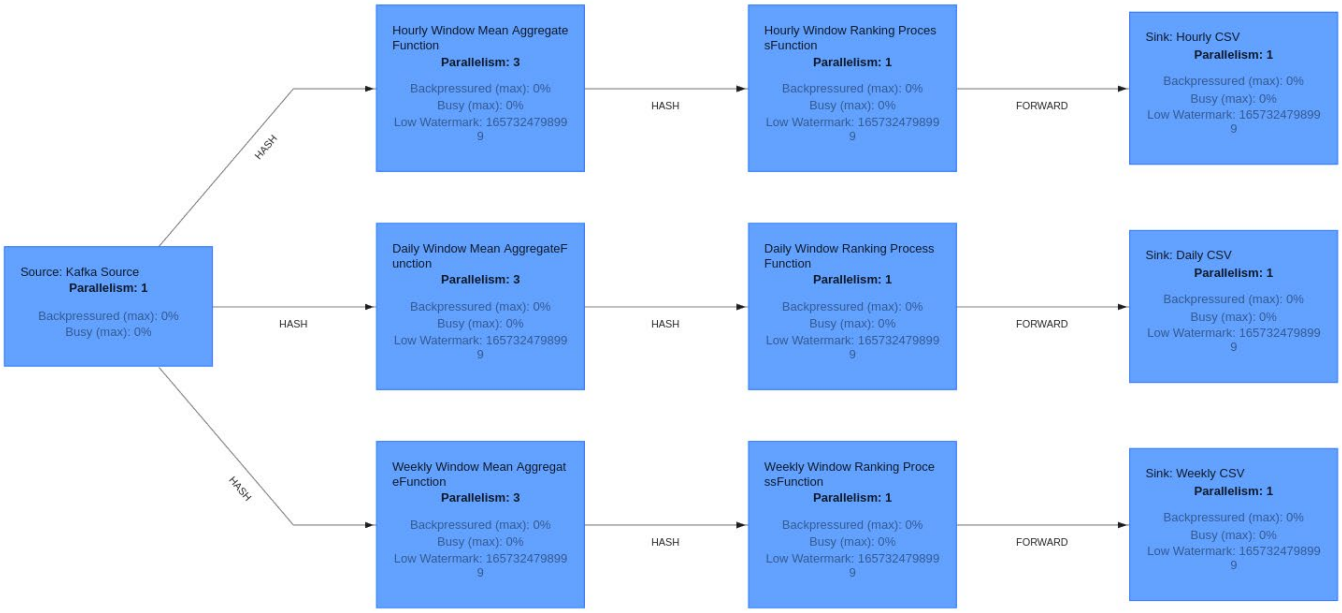
[1] https://hub.docker.com/r/confluentinc/cp-kafka
[2] https://hub.docker.com/r/obsidiandynamics/kafdrop
[3] https://hub.docker.com/_/flink
[4] https://hub.docker.com/r/confluentinc/cp-zookeeper
[5] https://stackoverflow.com/questions/60559935/apache-flink-job-is-not-scheduled-on-multiple-taskmanagers-in-kubernetes-replic
[6] https://stackoverflow.com/questions/53942004/apache-flink-number-of-taskmanagers-per-machine
[7] https://bit.ly/stackoverflow-tumbling-splitted
[8] https://cdn-shop.adafruit.com/datasheets/BST-BMP180-DS000-09.pdf
[9] https://nightlies.apache.org/flink/flink-docs-release-1.7/dev/connectors/kafka.html#the-deserializationschema
[10] https://bit.ly/mymaps-sabd

8. APPENDICE: EXECUTION GRAPHS

8.1. Query 1



8.2. Query 2



8.3. Query 3