Analisi in tempo reale di difetti nella produzione L-PBF con Apache Flink e Spark

Gentili Emanuele, Donnini Francesco*
emanuele.gentili@students.uniroma2.eu
Università di Roma Tor Vergata
Roma, Italia, Europa

ABSTRACT

Il progetto richiedeva la creazione di una pipeline per il processamento *real-time* di immagini prese durante la stampa di oggetti 3D usando la tecnologia Laser Powder Bed Fusion (L-BFS), una tecnologia di stampa 3D impiegata per realizzare parti in metallo a partire da un letto di polveri metalliche fuse per mezzo di un laser ad alta potenza e con un fascioconcentrato. L-PBF costruisce i componenti strato dopo strato, fondendo selettivamente polvere metallica tramite un laser, sulla base di un modello digitale (ad esempio, fornito tramite file CAD). I prodotti realizzati tramite L-BFS potrebbero essere soggetti a diversi tipi di difetti, dovuti a impuritá del materiale, instabilitá termiche o errori di calibrazione. Tradizionalmente, il controllo di qualitá nella L-PBF viene eseguito dopo la produzione, ma rilevare i difetti solo a posteriori comporta uno spreco di tempo, energia e risorse, poiché i pezzi difettosi devono essere scartati. Per ridurre queste inefficienze, possono essere utilizzate tecniche di monitoraggio e di rilevazione dei difetti in tempo reale. La tomografia ottica (OT) consente di catturare immagini termiche del letto di polvere a ogni strato del processo. Queste immagini mostrano la distribuzione delle temperature e, se analizzate correttamente, possono rivelare irregolaritá legate a potenziali difetti. L'analisi in tempo reale dei dati OT consente interventi immediati (per esempio, la sospensione del processo o la modifica dei parametri) per prevenire il degrado della qualitá.

1 INTRODUCTION

Lo scopo di questo progetto è rispondere ad alcune query su dati di monitoraggio del processo produttivo L-PBF, utilizzando l'approccio di processamento orientato ai data-stream con Apache Flink. In particolare, si richiede di effettuare un'analisi in tempo reale delle immagini di OT, cioè di istantanee termiche del letto di polvere durante il processo di stampa, una foto per ogni strato. Il dataset viene fornito attraverso un server REST, challenger, che viene utilizzato per simulare la generazione di dati in tempo reale e per valutare le performance delle query. Ogni foto viene divisa in settori della stessa dimensione denominati tile e il server fornisce una tile alla volta, ogni tile è identificato dai seguenti campi:

- `seq_id`: numero di sequenza unico per l'elemento di input;
- `print_id`: identificatore dell'oggetto in corso di stampa;
- `tile_id`: identificatore del tile all'interno del layer;
- `layer`: coordinata z del layer corrente;
- `tiff`: dati binari che rappresentano il tile in formato TIFF.

Le immagini rappresentano la distribuzione della temperatura sul letto di polvere per ogni strato dell'oggetto in fase di stampa. Ogni immagine codifica i valori di temperatura T_p per ogni punto $p = (x, y)$ del letto. Per ogni tile è richiesto di identificare i punti la cui temperatura è:

- < 5000 tali punti rappresentano aree di vuoto;
- > 65000 tali punti vengono considerati saturati e possono indicare la presenza di difetti.

Per la seconda query è richiesto il calcolo della deviazione di temperatura locale per ogni punto p, che viene definita come la differenza assoluta tra:

- la temperatura media dei vicini prossimi di un punto, cioè di tutti i punti con distanza di Manhattan $0 \leq d \leq 2$ da p, considerando i tre layer;
- la temperatura media dei vicini esterni, cioè di tutti i punti con distanza di Manhattan $2 < d \leq 4$ da p, considerando i tre layer.

La distanza di Manhattan, definisce formalmente, nel piano, la distanza L tra il punto $P_1 = (x_1, y_1)$ e il punto $P_2 = (x_2, y_2)$ come:

$$L(P_1, P_2) = |x_1 - x_2| + |y_1 - y_2|$$

Prima di essere spediti, i tile sono compressi nel formato MessagePack che è un formato binario per la serializzazione dei dati che permette di rappresentare coppie chiave-valore dove le chiavi sono stringhe e i valori possono essere di vari tipi come booleani, interi e stringhe oppure più complessi come liste o altri aggregati di coppie chiave-valore. Le prime due query, oltre che con Flink, sono state implementate anche con Spark Streaming. Tutti i risultati ottenuti sono stati validati con una versione sequenziale del programma.

2 BACKGROUND

2.1 Apache Flink

Apache Flink [1] è un framework e un motore di elaborazione distribuita per l'elaborazione con stato di flussi di dati limitati o illimitati. Flink supporta la definizione di operatori stateful, lo stato del task viene sempre mantenuto in memoria o, se la dimensione dello stato supera la memoria disponibile, su disco. Pertanto, i task eseguono tutti i calcoli accedendo allo stato locale, spesso in memoria, con latenze di elaborazione molto basse. Fintanto che le componenti esterne con cui Flink interagisce e prende i dati forniscano meccanismi di commit e rollback, Flink garantisce una consistenza dello stato exactly-once grazie a meccanimi di checkpointing periodico.

2.2 Apache Spark

Apache Spark [5] è un engine multilinguaggio per l'esecuzione di flussi di lavoro di data engineering, data science e machine learning su macchine a nodo singolo o cluster che supporta sia batch che

*Studenti magistrali di Ingegneria Informatica

stream processing. Spark include due engine per lo stream processing: Spark Streaming [6], che ormai è obsoleto, e il suo successore **Structured Streaming** [7]. Structured Streaming è un engine costruito al di sopra di SparkSQL, con l'obiettivo di essere scalabile e tollerante ai guasti. Structured Streaming consente di esprimere calcoli su flussi di dati (streaming computation) utilizzando la stessa semantica impiegata per elaborazioni batch. Il motore Spark SQL si occupa dell'esecuzione incrementale e continua delle query, aggiornando il risultato finale in tempo reale man mano che nuovi dati in streaming vengono ricevuti. L'API Dataset/DataFrame (disponibile in Scala, Java, Python e R) consente di modellare trasformazioni complesse su dati in streaming, incluse aggregazioni e finestre temporali basate su event-time. Structured Streaming garantisce la tolleranza ai guasti con semantica exactly-once end-to-end, grazie all'uso di checkpointing e Write-Ahead Logs (WAL). In sintesi, Structured Streaming offre un'infrastruttura per l'elaborazione di stream che è veloce, scalabile, fault-tolerant e che non richiede agli sviluppatori di gestire direttamente la complessità dello streaming. A livello implementativo, le query Structured Streaming sono, per impostazione predefinita, eseguite tramite un motore di micro-batch, che elabora i dati in streaming come una sequenza di piccoli batch.

2.3 Apache Kafka

Kafka [3] è un sistema distribuito composto da server e client che comunicano tramite un protocollo di rete TCP ad alte prestazioni. Può essere distribuito su hardware bare-metal, macchine virtuali e container in ambienti on-premise e cloud. Kafka combina tre elementi chiave per permettere di implementare i casi d'uso degli sviluppatori:

- publish e subscribe a flussi di eventi, inclusa l'importazione ed esportazione continua dei dati da altri sistemi;
- memorizzare flussi di eventi in modo durevole e affidabile per tutto il tempo desiderato;
- elaborare flussi di eventi man mano che si verificano.

Tutte queste funzionalità sono fornite in modo distribuito, altamente scalabile, elastico, fault-tolerant e sicuro.

3 SOLUTION DESIGN

La soluzione proposta [4] utilizza Flink per rispondere alle tre query e Spark per rispondere alle prime due. Questi framework vengono utilizzati per definire una pipeline costituita dalle seguenti fasi:

- analisi basata su soglia (Q1)
- windowing (Q2)
- analisi degli outliers (Q2)
- clustering degli outliers (Q3)

Flink legge i dati direttamente dal server grazie alle API `DataSource` [2], per quanto riguarda Spark, l'architettura diventa più complessa, perché si utilizza come sorgente dei dati Kafka che viene popolato in modo incrementale da un nodo che interagisce col challenger.

3.1 Apache Flink

In Flink per poter leggere i dati dal server sono state utilizzate le `DataSource` API che ci hanno permesso di recuperare i dati prodotti dal challenger e immetterli direttamente in Flink senza dover usare componenti di framework terzi. Le componenti fondamentali dell'architettura implementata per il processamento dei dati in Flink sono:

- Task Manager: eseguono i task definiti nel grafo diretto aciclico dell'applicazione Flink su uno stream di dati;
- Job Manager: ha diverse responsabilità legate al coordinamento dell'esecuzione distribuita delle applicazioni Flink: decide quando pianificare il task successivo (o un insieme di task), interviene in caso di task completati o errori durante l'esecuzione, coordina i checkpoint e il ripristino in caso di errori.

Possiamo avere sia più job manager che più task manager. Avere più job manager è utile in contesti in cui ci sono dei guasti ricorrenti e attraverso un algoritmo di elezione possiamo eleggere un nuovo master tra quelli in standby. Questo non è il caso in questione e quindi abbiamo deciso di usare un solo job manager.

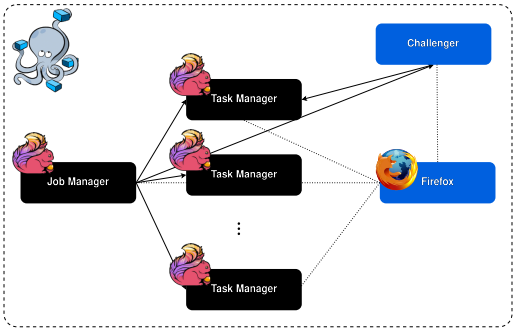

Figure 1: Ogni componente nella figura è un container Docker, il sistema viene eseguito tramite Docker Compose.

3.2 Apache Spark e Kafka

Per recuperare i dati dal server e immetterli su Spark è stato utilizzato Kafka. Un nodo apposito, denominato Producer, si occupa di interagire col challenger e pubblicare i batch su Kafka al topic `"benchmarks"`. Spark quando sarà pronto per la lettura dei dati si iscriverà a questo topic ed inizierà a consumare i dati all'interno della pipeline. In questo caso nell'architettura, oltre al server e alle componenti di Spark (master, worker e driver program) avremo anche il producer Kafka e i broker su cui vengono memorizzati i topic.

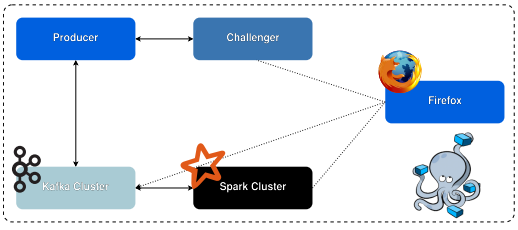

Figure 2: Il sistema viene eseguito tramite Docker Compose.

4 SOLUTION DETAILS

4.1 Pipeline Flink

4.1.1 Sorgente dei dati. La sorgente dei dati è stata implementata sfruttando le API `DataSource`, il cui componente fondamentale, date le nostre esigenze, é il `SourceReader` che può essere eseguito in parallelo in un task manager e si occupa di consumare i dati, in questo caso dal challenger, producendo uno o più stream di tuple, in questo caso batch. Il cuore di questo componente è la seguente funzione:

```
@Override
public InputStatus pollNext(ReaderOutput<Batch> readerOutput) throws Exception {
    var o = client.nextBatch(benchmark);
    if (o.isPresent()) {
        var entryTime = Instant.now();
        var batch = deserializer.deserialize(o.get());
        batch.setEntryTime(entryTime);
        readerOutput.collect(batch);
        return InputStatus.MORE_AVAILABLE;
    }
    return InputStatus.END_OF_INPUT;
}
```

Listing 1: Codice per la lettura dei dati dal challenger a Flink

Questa funzione viene chiamata ripetutamente fino a quando il server non ha più dati da inviare. Dalla specifica delle API del server viene indicato che quando non ci sono più batch da inviare, il server risponde con codice di errore 404, il client Java implementato per semplificare l'interazione col challenger ritorna un oggetto `Optional.empty` per segnalare a Flink che non ci sono più dati da consumare, a quel punto, Flink emette un oggetto speciale nel flusso che ne indica la fine. Durante l'operazione di deserializzazione, essendo necessario effettuare il parsing dell'immagine dal formato tiff a una matrice di interi, si coglie l'occasione per contare quanti di questi pixel superano la soglia di saturazione.

```
var count = 0L;
var image = ImageIO.read(new ByteArrayInputStream(tif));
var matrix = new int[image.getHeight()][image.getWidth()];
var data = image.getData();
for (var y = 0; y < image.getHeight(); y++) {
  for (var x = 0; x < image.getWidth(); x++) {
    var t = data.getSample(x, y, 0);
    matrix[y][x] = t;
    if (t > Thresholds.SATURATION) {
      count++;
    }
  }
}
return new TifWithSaturation(matrix, count);
```

Listing 2: Parsing dell'immagine TIF in un formato fruibile dall'applicazione flink e conteggio dei pixel oltre la soglia di saturazione.

Il flusso risultante composto da oggetti `Batch` viene diramato in due flussi, uno viene salvato su file e l'altro partizionato per chiave formata dalla coppia

$$(\texttt{print\_id}, \texttt{tile\_id})$$

In funzione della configurazione dell'applicazione, il flusso viene fornito a un operatore che usa una finestra basata sul tempo oppure una finestra implementata da noi.

4.1.2 Query 2. La seconda query richiede il processamento di tre immagini che abbiano la stessa chiave e livelli consecutivi. Per ogni tensore è necessario calcolarne gli outliers ossia i pixel la cui deviazione supera una certa soglia. Sono stati individuati due modi differenti per effettuare l'aggregazione di immagini in tensori:

- Una finestra scorrevole basata su event time, cioè l'istante in cui è stato prodotto l'evento dal challenger. In realtà non c'è un vero e proprio timestamp memorizzato nel batch quindi si considera come event time il layer dell'immagine convertito in secondi. Il layer è un candidato perfetto perché individua una relazione d'ordine tra tutte le immagini con la stessa chiave. La finestra ammette un tempo massimo di $10s$ per gli elementi fuori ordine.
- Una finestra implementata ad-hoc come operatore stateful che utilizza un meccanismo di doppio buffering per gestire l'arrivo di tuple fuori ordine.

Per indicare che il layer deve essere usato come misura di tempo dalla finestra scorrevole è necessario definire una strategia di watermarking:

```
WatermarkStrategy
  .<Batch>forMonotonousTimestamps()
  .withTimestampAssigner((TimestampAssignerSupplier<Batch>) context ->
    (TimestampAssigner<Batch>) (element, recordTimestamp) ->
      element.getLayer() * 1000L);
```

Listing 3: Definizione del campo layer del batch come event time.

Flink non garantisce comunque l'ordinamento tra le tuple all'interno di una finestra quindi è necessario effettuarne l'ordinamento prima di passare il tensore all'algoritmo che ne calcola gli outlier.

L'implementazione ad-hoc sfrutta due buffer che contengono esclusivamente elementi con la stessa chiave:

- `window`: una struttura dati `Deque<Batch>` ottimizzata per inserimenti/rimozioni dalla testa e dalla coda, particolarmente indicata per implementare le operazioni di scorrimento della finestra, un nuovo elemento viene inserito in coda e lo scorrimento viene effettuato eliminando dalla testa. Si ha la garanzia che ogni elemento in questo buffer sia nel giusto ordine: dato un qualsiasi elemento nel buffer di livello l, allora il predecessore ha come layer $l-1$ e il successore ha layer pari a $l+1$. Ogni sequenza di tre elementi in questo buffer è processabile.
- `buffer`: si occupa di memorizzare gli elementi arrivati fuori ordine. È una *min-priority queue* che usa il layer come chiave. Questa struttura dati rende particolarmente semplice ed efficiente il trasferimento degli elementi dal buffer alla finestra quando si verificano le giuste condizioni.

La finestra mantiene un altro attributo, `int startingOffset`, che indica qual è il layer minimo della prossima finestra da processare.

```
public boolean update(Batch b) {
  if (inorder(b)) {
    window.add(b);
    fillWindow();
    return true;
  }
  buffer.add(b);
  return false;
}
```

Listing 4: Procedura di aggiunta di un nuovo batch alla finestra.

Ogni volta che un nuovo batch arriva all'operatore, viene aggiunto alla finestra tramite il metodo `update()` che ritorna **`true`** se l'elemento è stato aggiunto alla finestra, **`false`** altrimenti. Un elemento `b` è in ordine se è vera una delle seguenti condizioni:

(1) la finestra è vuota e `b.getLayer() == startingOffset`, oppure
(2) la finestra non è vuota e il livello dell'ultimo batch (quello massimo all'interno della finestra) è l'immediato predecessore del livello di `b`.

Nel caso in cui si aggiunga un nuovo elemento alla finestra, potrebbe verificarsi che uno o più elementi nel buffer possano essere spostati alla finestra perché grazie al nuovo elemento vengono soddisfatte le invarianti della finestra. La procedura `fillWindow()` si occupa di effettuare questa operazione

```
private void fillWindow() {
  var i = window.getLast().getLayer() + 1;
  while (buffer.peek() != null && buffer.peek().getLayer() == i) {
    window.add(buffer.poll());
    i++;
  }
}
```

Listing 5: Procedura di trasferimento degli elementi, precedentemente fuori ordine, dal buffer di supporto alla finestra.

Una volta calcolati gli outliers, il flusso viene diramato a due operatori: uno è responsabile del ranking dei primi cinque outliers ordinati rispetto alla deviazione, mentre l'altro calcola i centri del cluster e quanti punti fanno parte di esso utilizzando l'algoritmo DBScan fornito dalla libreria `commons-math3` di Apache. Il ranking dei punti e i centri del cluster vengono salvati su file, inoltre quest'ultimi vengono inviati al challenger come richiesto dalla specifica.

4.2 La pipeline Spark

Come prima fase della pipeline, abbiamo l'acquisizione dei dati dal server. In Spark questa acquisizione avviene con l'ausilio di Apache Kafka. Il producer si occupa di creare un benchmark e avviare il client attraverso il quale attraverso delle API REST potrà richiedere al server di produrre i dati. Iterativamente viene chiamata l'API `next_batch()` che ritorna il batch successivo finché ce ne sono. É anche possibile specificare un numero massimo di batch che si possono generare. In quel caso la produzione dei batch si fermerà a tale limite.
I batch che vengono recuperati dal producer vengono pubblicati su un topic Kafka utilizzando il benchmark come chiave. Spark consumerà i dati prodotti comunicando col broker Kafka. I dati che vengono letti dal server e scritti in Kafka sono dei byte. Quando i dati arrivano a Spark esso deve deserializzarli per costruire l'oggetto `Batch` su cui dovrà eseguire le query. Proprio durante la fase di deserializzazione, viene analizzata l'immagine per individuare i punti saturati. Ovvero i punti la cui temperatura supera 65000. In questo modo si sta già rispondendo alla prima query senza dover ripetere la scansione dell'immagine e questa informazione verrà memorizzata nell'oggetto `Batch` deserializzato in modo da scriverla nel file di output.
La seconda query si concentra sul secondo e terzo stadio della pipeline. Per ogni tile si deve mantenere una finestra scorrevole degli ultimi 3 layer. L'implementazione della finestra è stata realizzata attraverso una `FlatMapGroupsWithStateFunction`. Dato che la finestra si deve mantenere per ogni tile, la chiave scelta per raggruppare gli elementi, come in Flink, è la coppia `(print_id,tile_id)`. Anche nel caso di Spark, non è garantito alcun ordinamento tra le tuple della finestra. Per far fronte a questo problema è stata utilizzata la stessa soluzione descritta per Flink. L'obiettivo della seconda query era quello di classificare come outliers i punti la cui deviazione di temperatura locale supera 6000. Quanto appena descritto è implementato all'interno della seguente funzione:

```
@Override
public Iterator<OutliersBatch> call(BatchKey key,
Iterator<Batch> values,
GroupState<KeyedBatchSlidingWindow> state) throws Exception {
  var window = getOrCreate(state);
  var batches = new ArrayList<OutliersBatch>();
  while (values.hasNext()) {
    var b = values.next();
    if (window.update(b)) {
      for (var w = window.getWindow(); !w.isEmpty(); w = window.getWindow()) {
        var filtered = w
          .stream()
          .filter(x -> x.getBatchId() >= 0).collect(Collectors.toList());
        batches.add(algorithm.compute(filtered));
      }
    }
  }
  state.update(window);
  return batches.iterator();
}
```

Listing 6: Procedura per il processamento delle tre immagini su Spark.

La logica utilizzata per l'identificazione degli outliers, così come la classe che implementa la finestra scorrevole col meccanismo di doppio buffering, è stata resa indipendente dal framework di processamento in modo da poterla riutilizzare in Flink e Spark senza dover riscrivere gli algoritmi. La suddetta funzione calcola tutti gli outliers. Verrà poi applicata un'altra funzione che ha il compito di calcolare la classifica dei 5 punti con deviazione di temperatura locale maggiore.

5 ESPERIMENTI

5.1 Apache Flink

Su Flink sono state provate due configurazioni:

- La versione che utilizza la finestra da noi proposta.
- La versione che utilizza la finestra basata su event time.

L'applicazione è stata eseguita su un cluster Flink costituito da un job manager e un numero di task manager pari a $n = 1, 2, 5$. I tempi sono riportati in secondi. È stata utilizzata la libreria NETEM per

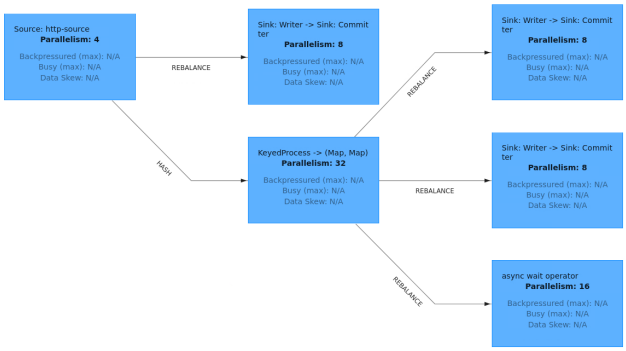

Figure 3: DAG dell'applicazione Flink che riporta il grado di parallelismo scelto per ogni operatore.

introdurre ritardi a livello di rete, tra i nodi del cluster, che seguono una distribuzione normale con parametri $\mu = 10ms$, $\sigma = 2ms$, inoltre il client applica un ritardo fisso di $100ms$ durante il trasferimento dei risultati al challenger. Per il calcolo del tempo di risposta media è stato raccolto il tempo d'ingresso di ogni batch nel sistema e il relativo tempo d'uscita (che nel nostro caso corrisponde al tempo in cui il batch processato viene scritto su file). Il calcolo del throughput per operatore ha richiesto la raccolta dei tempi d'ingresso di ogni batch nell'operatore, in particolare il throughput è stato calcolato come

$$X = \frac{N}{t_{\text{out},M} - t_{\text{in},m}}$$

dove N è il numero di batch, $t_{\text{out},M}$ è il tempo d'uscita dell'ultimo batch e $t_{\text{in},m}$ è il tempo d'ingresso del primo batch nell'operatore.

	R	σ	p_{99}	R_M	X
Q1	0.056	0.030	0.111	0.127	188.689
Q2	0.117	0.032	0.188	0.216	189.215
Q3	0.120	0.033	0.191	0.261	189.125

Table 1: Finestra ad-hoc, 1 worker.

	R	σ	p_{99}	R_M	X
Q1	0.058	0.029	0.112	0.183	182.769
Q2	0.127	0.032	0.188	0.286	184.436
Q3	0.128	0.032	0.190	0.292	184.587

Table 2: Finestra ad-hoc, 2 worker.

	R	σ	p_{99}	R_M	X
Q1	0.057	0.029	0.113	0.125	190.174
Q2	0.120	0.035	0.192	0.349	191.276
Q3	0.124	0.036	0.201	0.356	191.194

Table 3: Finestra ad-hoc, 5 worker.

n	X	R	p_{99}	R_M
1	138.46	3s230ms172μs	8s744ms988μs	9s343ms1μs
2	136.57	2s965ms37μs	8s331ms604μs	8s820ms281μs
5	137.15	3s329ms929μs	9s43ms943μs	9s764ms892μs

Table 4: Finestra ad-hoc, metriche raccolte dal micro-challenger.

	R	σ	p_{99}	R_M	X
Q1	0.062	0.030	0.117	0.204	150.000
Q2	0.320	0.079	0.516	0.583	152.484
Q3	0.325	0.083	0.522	0.582	152.465

Table 5: Finestra basata su event time, 1 worker.

	R	σ	p_{99}	R_M	X
Q1	0.060	0.030	0.114	0.156	158.751
Q2	0.311	0.079	0.507	0.593	159.964
Q3	0.315	0.080	0.523	0.605	159.908

Table 6: Finestra basata su event time, 2 worker.

	R	σ	p_{99}	R_M	X
Q1	0.061	0.030	0.115	0.168	158.158
Q2	0.302	0.082	0.503	0.572	159.426
Q3	0.307	0.083	0.509	0.600	159.426

Table 7: Finestra basata su event time, 5 worker.

n	X	R	p_{99}	R_M
1	133.29	1s340ms864μs	3s693ms998μs	4s142ms919μs
2	135.73	1s650ms993μs	4s677ms523μs	5s234ms734μs
5	136.11	1s614ms705μs	4s692ms951μs	5s423ms926μs

Table 8: Finestra basata su event time, metriche raccolte dal micro-challenger.

5.2 Spark

Spark è stato eseguito con un worker e tre worker.

	R	σ	p_{99}	R_M	X
Q1	0.006	0.0027	0.014	0.105	9.62
Q2	1.525	0.445	3.019	4.052	9.65

Table 9: 1 Worker.

	R	σ	p_{99}	R_M	X
Q1	0.011	0.023	0.066	0.551	9.15
Q2	0.976	0.979	2.945	22.741	22.22

Table 10: 3 Worker.

REFERENCES

[1] [n. d.]. Flink. https://flink.apache.org/what-is-flink/flink-architecture/.
[2] [n. d.]. Flink DataSource API. https://nightlies.apache.org/flink/flink-docs-master/docs/dev/datastream/sources/.
[3] [n. d.]. Kafka. https://kafka.apache.org/intro.
[4] [n. d.]. Repository del progetto. https://github.com/francescodonnini/progetto2.
[5] [n. d.]. Spark. https://spark.apache.org/.
[6] [n. d.]. Spark Streaming. https://spark.apache.org/docs/latest/streaming-programming-guide.html.
[7] [n. d.]. Structured Streaming. https://spark.apache.org/docs/latest/streaming/index.html.