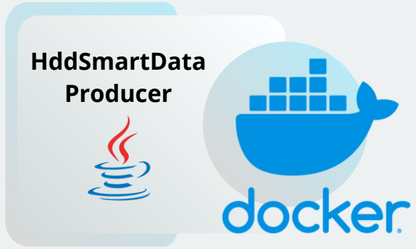
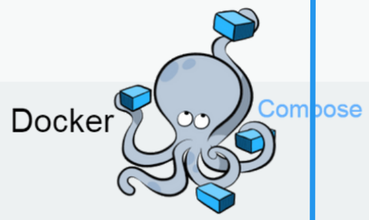
Analisi in tempo reale di eventi di monitoraggio di dischi con Apache Flink

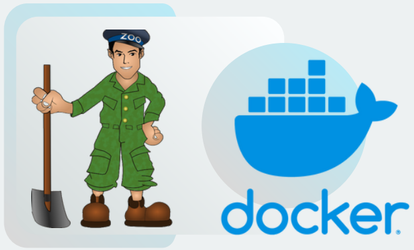
2023-2024

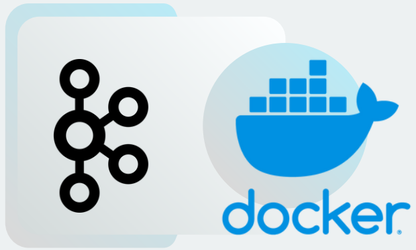
Marina Calcaura, Valerio Brauzi

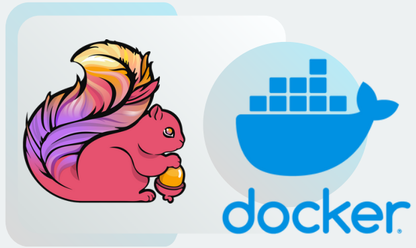
Architettura

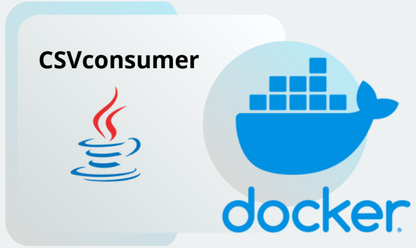




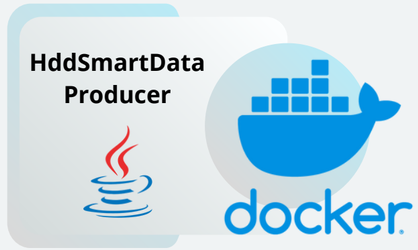








HddSmartDataProducer.java



- **Download del Dataset:** Se il file CSV non esiste localmente, scarica un file tar.gz contenente i dati SMART da un URL specificato.
- **Estrazione del Dataset:** Estrae il file tar.gz per ottenere il file CSV.
- **Configurazione Kafka:** Configura e crea il produttore Kafka.
- **Lettura e Invio Dati:** Legge il file CSV e invia i dati a Kafka su un topic denominato "input-data".

Viene simulato il passaggio del tempo per analizzare i dati come se fossero raccolti in tempo reale.
La simulazione si basa sulla differenza di tempo tra due record consecutivi.
Un fattore di velocizzazione rende la simulazione più veloce rispetto al tempo reale.

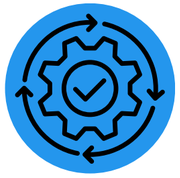
Query1

OBIETTIVO:

Per i vault con identificativo compreso tra 1000 e 1020, calcolare il numero di eventi, il valor medio e la deviazione standard della temperatura misurata sui suoi hard disk .

Consumer Kafka

Consuma i dati dal topic “input-data.”



Flink Stream Processing

Elabora i dati, li filtra, li aggrega e calcola le statistiche.

Producer Kafka

Pubblica i risultati aggregati su topic Kafka specifici.

Flink Stream Processing

Parsing e Filtraggio dei Dati:
I dati grezzi vengono trasformati in oggetti Record, che includono campi come timestamp, vaultId e temperatura. Vengono scartati i record non validi o incompleti.
I record vengono filtrati in base a criteri specifici, come l'intervallo richiesto di vaultId e la presenza di valori di temperatura non nulli.

Assegnazione dei Timestamp e Generazione dei Watermark

Aggregazione dei Dati per Finestra Temporale
I dati, raggruppati per vaultId, vengono aggregati in finestre temporali di 1 giorno, 3 giorni e in una finestra temporale che copre l'intero periodo dei dati disponibili.

```
// Extract timestamps and generate watermarks
DataStream<Record> parsedStream = stream
        .map(KafkaFlinkJob::parseRecord)
        .filter(record -> record != null)
        .filter(record -> record.getVaultId() >= 1000 && record.getVaultId() <= 1020)
        .filter(record -> record.getTemperature() != null)
        .assignTimestampsAndWatermarks(
                WatermarkStrategy.<Record>forBoundedOutOfOrderness(Duration.ofSeconds(30))
                        .withTimestampAssigner((event, timestamp) -> event.getTimestamp().getTime())
                        .withIdleness(Duration.ofMinutes(5))
        );
```

```
// Aggregate results in time windows of 1 day
DataStream<Tuple5<String, Integer, Long, Double, Double>> resultStream1Day = parsedStream
        .keyBy(Record::getVaultId) KeyedStream<Record, Integer>
        .window(TumblingEventTimeWindows.of(Time.days(1))) WindowedStream<Record, Integer, TimeWindow>
        .aggregate(new TemperatureStatisticsAggregator(), new WindowResultFunction()) SingleOut
        .map(new MetricRichMapFunction<>( windowId: "1 Day"));
```

Calcolo delle Statistiche con l'Algoritmo di Welford
Per ogni vaultId, viene creato un nuovo oggetto WelfordAccumulator, che tiene traccia della media e della deviazione standard dei dati di temperatura.

Alla fine della finestra temporale, l'accumulatore contiene le statistiche aggregate (media, deviazione standard, conteggio degli eventi) per quel vaultId.

Emissione dei Risultati Aggregati

```java
@Override
public Statistics getResult(WelfordAccumulator accumulator) {
    Statistics stats = new Statistics();
    stats.setVaultId(accumulator.getVaultId());
    stats.setEventCount(accumulator.getCount());
    stats.setMeanTemperature(accumulator.getMean());
    stats.setStdDevTemperature(accumulator.getStdDev());
    return stats;
}
```

```java
Iterator<Statistics> iterator = input.iterator();
if (!iterator.hasNext()) {
    System.err.println("L'iterabile di input è vuoto");
    return;
}

while (iterator.hasNext()) {
    Statistics statistics = iterator.next();
    if (statistics != null) {
        out.collect(new Tuple5<>(
                windowStart,
                statistics.getVaultId(),
                statistics.getEventCount(),
                statistics.getMeanTemperature(),
                statistics.getStdDevTemperature()
        ));

    } else {
        System.err.println("L'oggetto Statistics è null");
    }
```

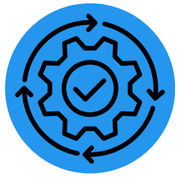
Query2

OBIETTIVO:

Calcolare la classifica aggiornata in tempo reale dei 10 vault che registrano il piu alto numero di fallimenti nella stessa giornata, riportando per ogni vaultId il numero di fallimenti, il modello e numero seriale degli hard disk guasti

Consumer Kafka

Consuma i dati dal topic “input-data.”



Flink Stream Processing

Elabora i dati, li filtra, li aggrega e calcola le statistiche.

Producer Kafka

Pubblica i risultati aggregati su topic Kafka specifici.

Flink Stream Processing

Parsing e Filtraggio dei Dati:
I dati grezzi vengono trasformati in oggetti Record, che includono campi come timestamp, vaultId e failure. I record non validi vengono scartati.
I record vengono filtrati per assicurarsi che solo quelli con fallimenti (failure == 1) siano processati ulteriormente

Assegnazione dei Timestamp e Generazione dei Watermark

Aggregazione dei Dati per Finestra Temporale
I dati, raggruppati per vaultId, vengono aggregati in finestre temporali di 1 giorno, 3 giorni e in una finestra temporale che copre l'intero periodo dei dati disponibili.

```
// Extract timestamps and generate watermarks
DataStream<Record> parsedStream = stream
        .map(Query2::parseRecord)
        .filter(record -> record != null)
        .filter(record -> record.getFailure() == 1) // Ensure only records with failures are processed
        .assignTimestampsAndWatermarks(
                WatermarkStrategy.<~>forBoundedOutOfOrderness(Duration.ofSeconds(30))
                        .withTimestampAssigner((event, timestamp) -> event.getTimestamp().getTime())
                        .withIdleness(Duration.ofMinutes(5))

        );
```

```
// Aggregate failures per vault in time windows of 1 day
DataStream<Tuple3<Integer, Integer, List<DiskFailure>>> failureStream1Day = parsedStream
        .keyBy(Record::getVaultId) KeyedStream<Record, Integer>
        .window(TumblingEventTimeWindows.of(Time.days(1))) WindowedStream<Record, Integer, TimeWindow>
        .aggregate(new FailureAggregator()) SingleOutputStreamOperator<Tuple3<Integer, Integer, List<DiskFailure>>>
        .map(new MetricRichMapFunction<>( windowId: "1 Day"));
```

Aggregatore dei fallimenti

Per ogni finestra temporale, viene creato un nuovo accumulatore.
Ogni record di fallimento viene aggiunto all'accumulatore, aggiornando il conteggio dei fallimenti e la lista dei dettagli dei dischi.
Alla fine della finestra temporale, l'accumulatore contiene il numero totale di fallimenti e i dettagli dei dischi per quel vaultId

Calcolo dei Top 10 Vault

Si utilizza una finestra globale (windowAll) per raccogliere tutti i risultati aggregati di una determinata finestra temporale.
I risultati vengono ordinati in base al numero di fallimenti (in ordine decrescente).
Vengono selezionati i primi 10 record dalla lista ordinata.

```
@Override
public Tuple3<Integer, Integer, List<DiskFailure>> createAccumulator() {
    Tuple3<Integer, Integer, List<DiskFailure>> accumulator = new Tuple3<>(0, 0, new ArrayList<>());
    //System.out.println("Created new accumulator: " + accumulator);
    return accumulator;
}

@Override
public Tuple3<Integer, Integer, List<DiskFailure>> add(Record value, Tuple3<Integer, Integer, List<DiskFailure>> accumulator) {
    //System.out.println("Adding value to accumulator: " + value);
    accumulator.f0 = value.getVaultId();
    accumulator.f1 += value.getFailure();
    accumulator.f2.addAll(value.getDiskFailures());
    //System.out.println("Updated accumulator: " + accumulator);
    return accumulator;
}
```

```
vate static DataStream<String> calculateTop10(DataStream<Tuple3<Integer, Integer, List<DiskFailure>>> failureStream, Time wind

return failureStream
        .windowAll(TumblingEventTimeWindows.of(windowSize, offset))
        .apply(new AllWindowFunction<Tuple3<Integer, Integer, List<DiskFailure>>, String, TimeWindow>() {
            @Override
            public void apply(TimeWindow window, Iterable<Tuple3<Integer, Integer, List<DiskFailure>>> values, Collector<Stri
                List<Tuple3<Integer, Integer, List<DiskFailure>>> sortedFailures = StreamSupport.stream(values.spliterator(),
                        .sorted((a, b) -> Integer.compare(b.f1, a.f1))
                        .limit( maxSize: 10)
                        .collect(Collectors.toList());
```

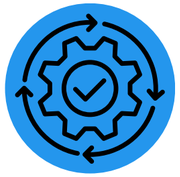
Query3

OBIETTIVO:

Calcolare il minimo, 25-esimo, 50-esimo, 75-esimo percentile e massimo delle ore di funzionamento degli hark disk per i vault con identificativo tra 1090 (compreso) e 1120 (compreso).

Consumer Kafka

Consuma i dati dal topic “input-data.”



Flink Stream Processing

Elabora i dati, li filtra, li aggrega e calcola le statistiche.

Producer Kafka

Pubblica i risultati aggregati su topic Kafka specifici.

Flink Stream Processing

Parsing e Filtraggio dei Dati:
I dati grezzi vengono trasformati in oggetti PowerOnHoursRecord, ciascuno contenente campi come timestamp, vaultId, serial e powerOnHours.
I record non validi vengono scartati.
I record vengono filtrati per assicurarsi che solo quelli con vaultId compresi tra 1090 e 1120 siano processati ulteriormente.

Assegnazione dei Timestamp e Generazione dei Watermark

Aggregazione dei Dati per Finestra Temporale
I dati, raggruppati per vaultId, vengono aggregati in finestre temporali di 1 giorno, 3 giorni e in una finestra temporale che copre l'intero periodo dei dati disponibili.

```
DataStream<PowerOnHoursRecord> recordsStream = stream
        .map(Query3::parseRecord)
        .filter(Objects::nonNull)
        .filter(record -> record.getVaultId() >= 1090 && record.getVaultId() <= 1120)
        .assignTimestampsAndWatermarks(
                WatermarkStrategy.<PowerOnHoursRecord>forBoundedOutOfOrderness(Duration.ofSeconds(30))
                        .withTimestampAssigner((event, timestamp) -> event.getTimestamp().getTime())
        );
```

```
DataStream<Tuple8<String, Integer, Long, Double, Double, Double, Double, Double>> results1Day = recordsStream
        .keyBy(PowerOnHoursRecord::getVaultId) KeyedStream<PowerOnHoursRecord, Integer>
        .window(TumblingEventTimeWindows.of(Time.days(1))) WindowedStream<PowerOnHoursRecord, Integer, TimeWindow>
        .aggregate(new LatestPowerOnHoursAggregator(), new WindowResultFunction()) SingleOutputStreamOperator<Tuple8<S
        .map(new MetricRichMapFunction<>( windowId: "1 Day"));
```

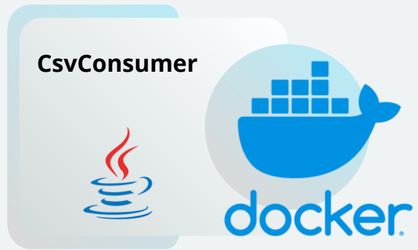
Aggregatore per i record più recenti
Dato che il valore di powerOnHours è cumulativo nel dataset, l'aggregatore *LatestPowerOnHoursAggregator* ha il compito di mantenere il record di tempo di accensione (powerOnHours) più recente per ciascun disco, identificato dal serial_number, all'interno di una finestra temporale definita.

Emissione dei Risultati
Ottenuta la lista dei record più recenti per ciascun disco alla fine della finestra temporale, si utilizza TDigest per calcolare le statistiche richieste.
I risultati calcolati vengono emessi come una tupla contenente il timestamp di inizio finestra, vaultId, conteggio dei record, e le statistiche calcolate

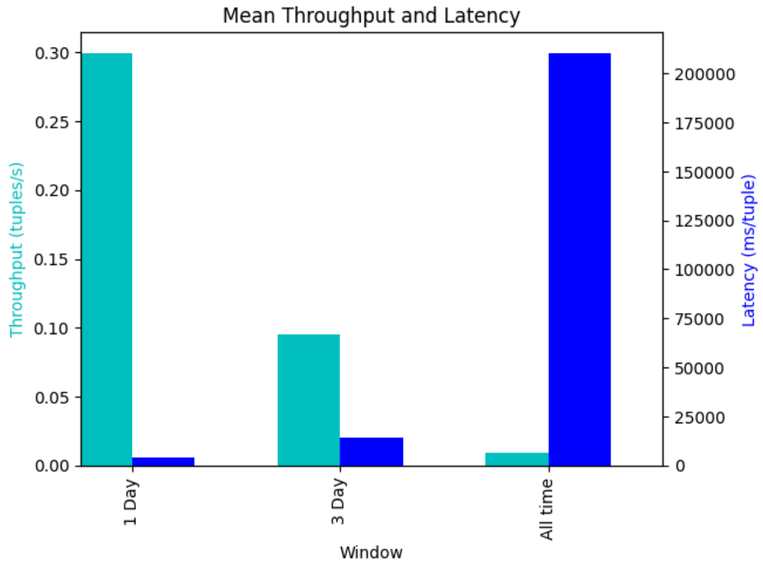
```
public static class WindowResultFunction implements WindowFunction<List<PowerOnHoursRecord>, Tuple8<String, Integer, Long
    1 usage
    private static final DateTimeFormatter formatter = DateTimeFormatter.ofPattern("yyyy-MM-dd HH:mm:ss").withZone(ZoneId
    @Override
    public void apply(Integer key, TimeWindow window, Iterable<List<PowerOnHoursRecord>> input, Collector<Tuple8<String,
        List<PowerOnHoursRecord> records = input.iterator().next();

        TDigest digest = TDigest.createDigest( compression: 100.0);
        for (PowerOnHoursRecord record : records) {
            digest.add(record.getPowerOnHours());
        }

        double min = digest.quantile( v: 0.0);
        double p25 = digest.quantile( v: 0.25);
        double p50 = digest.quantile( v: 0.50);
        double p75 = digest.quantile( v: 0.75);
        double max = digest.quantile( v: 1.0);
        long count = digest.size();

        String windowStart = formatter.format(Instant.ofEpochMilli(window.getStart()));

        out.collect(new Tuple8<>(windowStart, key, count, min, p25, p50, p75, max));
    }
```

CsvConsumer.java

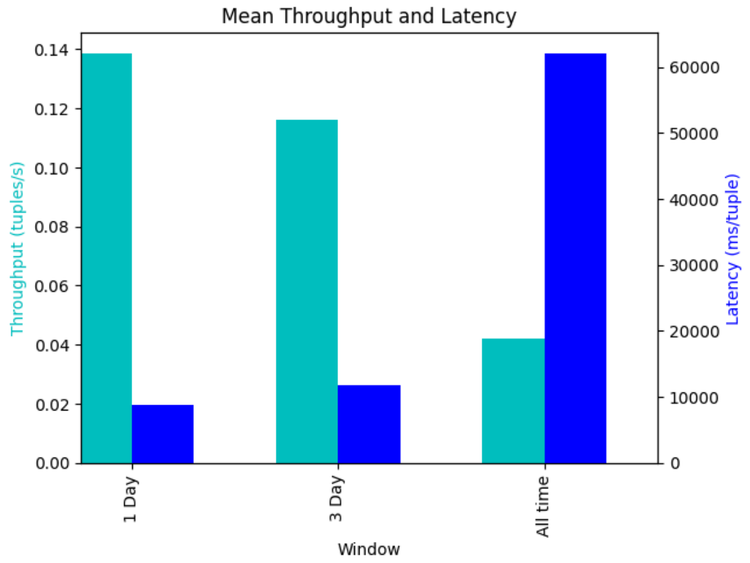
- Consuma messaggi dai topic Kafka.
- Scrive i dati in file .CSV



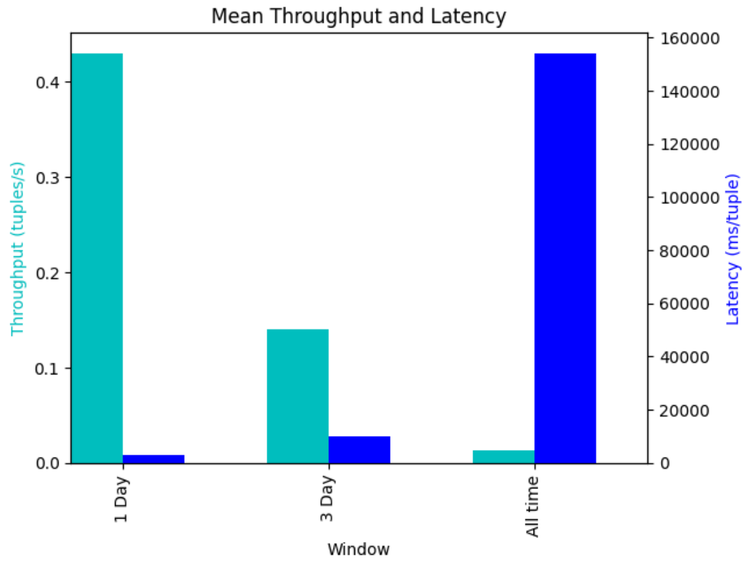
Valutazione delle Performance



Query1



Query2



Query3

Grazie per l’attenzione!