**ALMA MATER STUDIORUM**
**UNIVERSITÀ DEGLI STUDI DI BOLOGNA**

---

SCUOLA DI INGEGNERIA E ARCHITETTURA

Dipartimento di Informatica — Scienza e Ingegneria

Corso di Laurea in Ingegneria Informatica Magistrale

# Progettazione e Sviluppo di Applicazioni di Stream Processing su Apache Kafka

Tesi di Laurea in Ingegneria Informatica

Tesi di Laurea di:

Paolo Verdini

Relatore:
Chiar.mo Prof. Ing. Paolo Bellavista

Correlatore:
Dott. Ing. Luca Foschini

# Introduzione

Al giorno d'oggi, le aziende hanno sempre più flussi di dati generati continuamente da ogni tipo di fonte e per ogni tipo di motivazione: che sia per fini di business, o per fini di controllo e manutenzione. L'elaborazione real-time delle informazioni contenute in tali flussi di dati è spesso un potenziale vantaggio, sia a livello competitivo che in termini di efficienza ed efficacia dei processi aziendali (ottimizzazione dei costi, riduzione dei rischi, soddisfacimento dei clienti, incremento della qualità…).

Spesso, se si pensa al concetto di streaming ci si rimanda a insiemi di dati e informazioni generati, ad esempio, da sensori disponibili grazie all'IoT, alle app su dispositivi mobili, da web clickstream o da dati transazionali. In generale, le informazioni contenute in questi stream di dati sono rilevanti per le aziende nel momento in cui permettono di creare valore (arricchimento o miglioramento funzionalità) o di limitare un danno (economico).

Alcuni esempi di scenario applicativo che consentono di esplicitare le potenzialità del data streaming possono essere:

- Manutenzione preventiva: evento generato da un sensore significante un comportamento anomalo di un'attrezzatura o macchina può anticipare un intervento di manutenzione che evita il blocco dell'attrezzatura o macchina stessa.
- Efficienza ed efficacia di processo: il monitoraggio di un flusso di produzione può ottimizzare l'efficienza del processo e aumentare la qualità del prodotto finale.
- Incremento dei volumi di business: il clickstream di navigazione di un potenziale cliente su un sito web abilita la proposizione di ulteriori opzioni d'acquisto o alternative in caso di non disponibilità.
- Individuazione di frodi: successione di eventi "inserimento pin" troppo ravvicinati e per un tempo anomalo può essere sintomo di una potenziale frode.

La cosa che si evince considerando questi esempi è che, con il passare del tempo impiegato ad estrarre le informazioni contenute in questi dati, il valore correlato decade. Al fatto di riuscire ad analizzare ed estrarre le informazioni (spesso) in tempi estremamente rapidi, si aggiunge la complessità, in quanto i flussi in questione sono caratterizzati da enormi volumi di dati, spesso eterogenei: si parla di Big Data.

Lo stream processing & analytics si applica proprio dove è necessario estrarre informazioni da un flusso di Big Data in un tempo inferiore a una certa soglia: sulla base di questi analytics, si possono spesso prendere decisioni quasi in tempo reale, si può essere allertati al verificarsi di potenziali eventi negativi o si può essere supportati da processi automatizzati piuttosto che avere interazione umana (spesso più lunga e dispendiosa da attuare).

Il lavoro di questa tesi mira proprio ad implementare un servizio di analytics sfruttando tutte le funzionalità e potenzialità dello (event) stream processing di Big Data, derivanti da sensori che effettuano transazioni in tratti autostradali, per andare a capire l'andamento di tali transazioni, i fallimenti e i potenziali rischi che possono esserci sul guasto dei sensori e delle OBU, ovvero le On Board Unit (device che emettono i segnali catturati dai sensori – per intendersi, un esempio potrebbe essere un meccanismo come il Telepass a un casello autostradale).

Il primo capitolo introduce i concetti di Big Data e Stream Processing, insieme a tutte le principali tecnologie attualmente disponibili sul mercato per implementare tali concetti.

Il secondo capitolo verte sull'attuale stato dell'arte relativo al progetto della tesi, quali soluzioni già esistenti o solo progettate.

Il terzo capitolo descrive tutte le principali tecnologie utilizzare per lo sviluppo del progetto in questione. Ci sarà un particolare focus sulla piattaforma di streaming Apache Kafka (con tutti i relativi moduli e librerie) e su Spring (in particolare la parte delle REST API e Spring Kafka).

Il quarto capitolo mostra la fase di configurazione degli strumenti per il progetto, le funzionalità da implementare, l'architettura e le particolari scelte effettuate in fase di analisi.

Il quinto capitolo descrive il lavoro svolto, dei dettagli di implementazione e i risultati sperimentali su efficienza, oltre a una parte di confronto con le metodologie adottate precedentemente a questo progetto.

Infine, nel quinto capitolo vengono stilate le conclusioni, in cui si effettuano le valutazioni finali, si mostrano i risultati ottenuti, i punti a favore ed eventuali criticità, concludendo con possibili sviluppi futuri.

# 1. Big Data e Stream Processing

## 1.1 Introduzione Big Data

Con Big Data si intende mostrare una piattaforma che fornisce una soluzione a componenti per la gestione dinamica di Big Data, che non sono solo puri dati, ma anche processamenti (online e offline). Il primo elemento fondamentale dei Big Data è analizzare cosa siano veramente. Anche quando si ha una grande quantità di dati, non è detto che si parli di Big Data: un file Excel lungo o un DB sono esempi di grandi moli di dati che non necessariamente sono Big Data. Per capire cosa contraddistingue un Big Data da un qualunque altro grande volume di dati, si devono analizzare le cosiddette "cinque V", alle quali, spesso, viene aggiunta una sesta (correlata con tutte le altre cinque):

- Volume: ci deve essere una quantità considerevole di dati.
- Varietà: i dati devono essere vari, provenire da sorgenti eterogenee anche molto diverse tra loro (diverse in formato, in accuratezza, precisione …). Alcune fonti possono essere strutturate, come per esempio campi di un DB, altre possono non esserlo, come i post di Facebook.
- Velocità: i dati vengono prodotti con data rate molto alti, spesso così alti che non si riesce, o non è economicamente conveniente, memorizzarli tutti in memoria persistente. Negli scenari tradizionali di Big Data, i dati vengono tutti salvati in storage e analizzati a posteriori, in batch: ma molto spesso non è questo quello che vogliamo fare nei Big Data. Per esempio, certi dati provengono dal monitoraggio di automobili connesse tra loro, che generano svariati gigabyte di dati al secondo ciascuna. Non avrebbe senso, per esempio, analizzare dei dati riguardanti traffico in una certa zona di una città il giorno dopo, perché l'informazione che se ne ricava sarebbe servita in "near" real time, per mandare un eventuale segnale alle altre automobili per non farle incappare in un potenziale ingorgo.
- Valore: estrazione di valore dai dati, ovvero dagli svariati gigabyte che arrivano dai sensori, si devono estrarre solo le informazioni che interessano; questo comporta estrazione di informazioni di alto livello da dati di basso livello.
- Veridicità: in scenari tradizionali, si è abituati a dati tutti veritieri, con lo stesso livello di qualità in senso di affidabilità; in scenari Big Data, spesso i dati che vengono usati hanno livelli di qualità molto differenziati tra di loro. Si può addirittura incorrere in dati falsi iniettati

da competitor, per cercare di trascinare in estrazione di dati fasulli, con scopi personali. Non è possibile fidarsi completamente dei dati in questi scenari.

Spesso si parla della sesta V, ovvero la Variabilità: questa evidenzia il fatto che i Big Data sono una fluttuazione dinamica importante e fa riferimento a qualunque delle cinque V. Un esempio potrebbe essere la variabilità della velocità: in alcuni istanti si potrebbe avere un data rate molto alto, in altri molto basso, a seconda delle circostanze.

Si parla di Big Data ogni volta che viene fatta una ricerca su un motore, come Google, oppure quando si prendono in considerazione tutti i post su Facebook ma ancora quando si hanno scenari di Smart City e le informazioni dei sensori vanno raccolte e analizzate. L'ampio ventaglio di contesti in cui i Big Data possono essere utilizzati ha portato alla nascita di molte specifiche soluzioni, applicabili in ogni tipo di situazione: Correlation sui dati, Stream processing, Visual Analytics, Real-Time Analytics, Sicurezza e QoS, nuove architetture hardware e di storage.

Si potrebbe pensare allo stock market: se si hanno buone tecniche di Big Data, si possono osservare le azioni del mercato nei mesi o anni precedenti e cercare di predire l'andamento futuro, basandosi su cosa ha influenzato l'andamento passato. Allo stesso modo, si hanno i sistemi di sorveglianza: non sono dati strutturati, ma flussi di byte che devono essere analizzati in near real time, come leggere la targa di un'auto per identificare quelle rubate, cogliere la faccia di un malvivente e così via. Ancora, si hanno realtà applicative anche distanti dall'ambito prettamente informatico: le smart grid. L'energia elettrica è prodotta in gran parte dalle centrali elettriche tradizionali, ma cosa succede dal momento in cui introduciamo elettricità ricavata da fonti rinnovabili? L'energia elettrica non è conservabile, deve essere prodotta seguendo il profilo di consumo quotidiano di una certa zona coperta. Le fonti rinnovabili non garantiscono una continua generazione di corrente elettrica: è necessario analizzare lo storico dei consumi, il tempo, le condizioni di luce e altre informazioni al fine di tenere sotto controllo i profili di elettricità per non mandare in blackout interi quartieri: questo lavoro viene fatto dai Big Data.

## 1.2 Architettura

Il mondo dei Big Data è così rilevante che tante organizzazioni di standardizzazione hanno iniziato a ragionare su come deve essere fatto dal punto di vista standard un'architettura generale di soluzione ai problemi di Big Data. Una delle più rilevanti è lo sforzo del NIST (National Institute of Standards

and Technology). Il NIST ha standardizzato alcuni termini di Cloud Computing (pubblico, privato, ibrido) e tante altre terminologie, ormai accettate universalmente. L'architettura che il NIST propone in ambito di Big Data è quella riportata in figura 1.

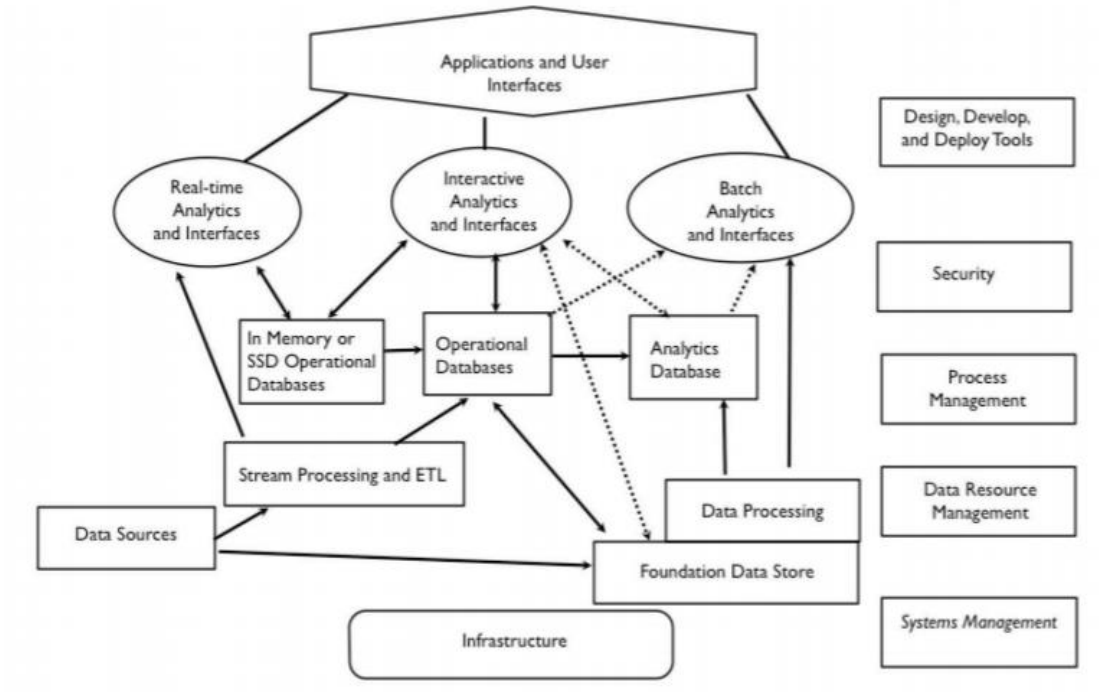


Figura 1. Architettura Big Data proposta da NIST

Il concetto più importante di questa architettura è la distinzione, subito prima delle applicazioni e interfaccia utente, in tre tipologie di Big Data; queste tipologie sono così diverse da aver bisogno al di sotto di meccanismi e strumenti significativamente diversi.

- Batch: piattaforma di Big Data utile in scenari applicativi in cui non si hanno bisogno di risultati di analytics in real time, ma in cui è accettabile che i dati prodotti vengano memorizzati da qualche parte e, in futuro più o meno prossimo, vengano fatte delle analisi su di essi. Si parla di analisi post-mortem dei dati di monitoraggio. Un esempio di applicazione di questo tipo di piattaforma è nella gestione dei log: se dovesse succedere qualcosa di strano, sarebbe possibile andare ad analizzare i log a posteriori, anche dopo giorni/settimane/mesi/anni.
- Real Time: piattaforma utile in applicazioni che necessitano di analisi in tempo reale sotto vincoli di latenza. In scenari di questo genere, i risultati di Analytics devono arrivare in tempo utile per riuscire a fare determinate operazioni: se arrivano tardi, molto probabilmente non servono nemmeno più. In Real Time Analytics, secondo NIST possono anche esserci supporti di memorizzazione, per esempio DB di analytics o DB operazionale di dati, che però non possono essere utilizzati durante l'analisi, in quanto non c'è abbastanza tempo per farlo.

- Interactive Analytics: categoria un po' centrale rispetto alle altre due, si basa su un'analisi post mortem dei dati già raccolti, ma il cui tipo di funzioni analitiche di analisi dei dati è modificabile runtime dall'utente. Quest'ultimo può, quindi, interagire con il sistema per cambiare quello che vuole osservare nei dati, prevedendo che il suo cambiamento di analytics produca dopo poco anche i risultati corrispondenti.

# 1.3 Stream Processing

Focalizzando l'attenzione sulla classificazione di piattaforma Real Time di Big Data del NIST, la pratica di effettuare operazioni su una enorme mole di dati in tempo reale è lo Stream Processing. Lo Stream Processing è nato come processamento real time di dati con una certa frequenza: oggigiorno, il workflow di uno Stream Processing può essere associato a una pipeline, in cui si ha la generazione dei dati, il processamento degli stessi e la consegna in un luogo finale (un DB o qualunque altra cosa).

## 1.3.1 Architettura e funzionalità

Le operazioni che lo Stream Processing effettua sui dati possono essere molteplici: da aggregazioni (somma, media o deviazione standard), passando per analytics (predizione di un evento futuro analizzando pattern nei dati), arricchimento (combinare i dati con altre sorgenti dati per creare più informazioni) e fare ingestion (inserire dati in un DB).

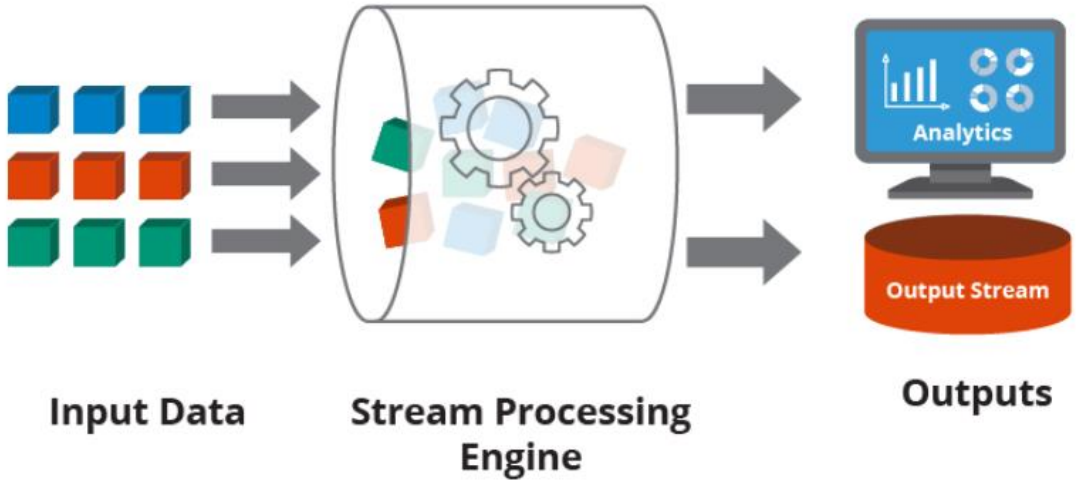


Figura 2. Workflow Stream Processing

Al contrario del Batch Processing, il quale era un'operazione solitamente schedulata periodicamente o attraverso dei particolari "eventi" (volume che raggiunge una certa soglia di dimensioni), lo Stream Processing consente alle applicazioni di rispondere a nuovi eventi nel momento (o quasi) in cui si verificano: piuttosto che raggruppare i dati e fare operazioni periodicamente, si analizzano e processano i dati immediatamente, appena generati.

Lo Stream processing è molto spesso utilizzato in scenari in cui i dati sono generati come una serie di eventi, come dati provenienti da sensori IoT, transazioni di pagamenti, log di server o applicazioni. Il paradigma comune prevede una struttura del tipo pub/sub e source/sink. I dati e gli eventi sono generati da un publisher o un source e consegnati all'applicazione di stream processing, dove i dati stessi sono processati con qualunque tipo di operazioni; successivamente, i risultati dei processamenti sono mandati a un subscriber o un sink.

## 1.3.2 Casi d'uso Stream Processing

I tipici casi d'uso dello Stream Processing sono situazioni in cui accadono degli eventi e sono necessarie azioni di risposta da effettuare immediatamente in risposta a tali eventi. I più comuni sono:

- Riconoscimento Real-Time di frodi o anomalie: adesso, con lo stream processing, al solo swipe di una carta di credito e immissione del pin, è possibile lanciare algoritmi di riconoscimento di frodi in real-time, senza dover attendere, sia da parte del consumatore che il negoziante.

- IoT edge analytics: un esempio di Stream Processing in IoT è quello dell'Industrial IoT, nell'infrastruttura manufatturiera. Qui è possibile identificare anomalie in macchinari che possono identificare problemi da risolvere e ottimizzare operazioni di qualunque tipo. Con un real time Stream Processing un operaio può riconoscere che una certa catena di produzione sta avendo troppe anomalie: è possibile quindi fermare il tutto prima che guasti gravi si verifichino, intervenendo dove necessario.
- Real Time marketing e advertising personalizzate: con lo Stream Processing, le aziende possono inviare pubblicità e offerte personalizzate ai consumatori, sia sconti per qualcosa aggiunto a un carrello di un sito e-commerce, che eventuali pubblicità di prodotti affini a quelli già visualizzati o cercati.

## 1.4 Event Stream Processing

L'Event Stream Processing è l'approccio con cui si effettua processamento su una serie di dati che sono originati da un sistema che li crea continuamente (potenzialmente all'infinito). Il termine "event" qua si riferisce a ogni dato/informazione nel sistema, mentre lo "stream" è l'effettivo flusso di questi eventi da una sorgente a una destinazione. Le azioni di processamento che vengono effettuate su questi dati sono le stesse menzionate per lo Stream Processing.

### 1.4.1 Architettura

Event Stream Processing è spesso interpretata come il complementare del Batch Processing: quest'ultimo, come già detto in precedenza, effettua processamenti in un grande insieme di dati statici (post mortem), mentre l'Event Stream Processing è necessaria per situazioni in cui le azioni devono essere eseguite il più presto possibile.

Il funzionamento dell'Event Stream Processing è concettualmente semplice: anziché vedere i dati come un intero insieme, li processa come se fossero un continuo flusso infinito; questo introduce la necessità di nuove tecnologie ad hoc.

Negli Event Stream Processing, sono necessari due principali tipologie di tecnologie: un sistema che memorizza gli eventi e una tecnologia che aiuta gli sviluppatori a scrivere applicazioni che

consumano gli eventi ed effettuano i processamenti. Il primo componente è simile a un data storage e memorizza i dati basandosi su timestamp: per esempio, è possibile effettuare misurazioni della temperatura esterna ogni minuto del giorno e trattare tali misurazioni come un event stream. In questo caso, ogni evento è la misurazione della temperatura con insieme il timestamp di quando tale lettura è stata effettuata. Spesso questa parte è gestita da Apache Kafka, di cui si parlerà successivamente. Il secondo componente è effettivamente l'Event Stream Processing che permette di effettuare i processamenti sui dati. Sebbene molti Stream Processor siano molto simili tra loro, quelli basati su "in-memory" Processing sono molto veloci a processare grandissime quantità di dati. Un esempio è Hazelcast Jet.

### 1.4.2 Casi d'uso

In realtà, i casi d'uso sono praticamente gli stessi dello Stream Processing, in questo l'Event Stream Processing ne è solo una variante. Infatti, gli scenari d'uso sono il processamento di pagamenti, il rilevamento di frodi e di anomalie, manutenzione predittiva e IoT Analytics.

L'Event Stream Processing è anche molto efficace in situazioni in cui la granularità dei dati è critica. Per esempio, basta pensare al fatto che il cambiamento del prezzo di un prodotto è più importante, per un trader, del prezzo in sé e questo tipo di Processing permette di tenere traccia di tutti i cambiamenti e prendere la migliore decisione di trading.

Un ulteriore scenario è quello del CDC (Change Data Capture), in cui ogni cambi individuali di un DB sono memorizzati. In questi casi, è possibile utilizzare questi cambi memorizzati per identificare eventuali pattern di utilizzo che possono aiutare a definire strategie di ottimizzazione.

## 1.5 Tecnologie di Stream Processing

In questo capitolo verranno illustrate brevemente le principali tecnologie esistenti per implementare uno Stream processing, comparandole e spiegandone i punti di forza e le debolezze.

### 1.5.1 Aspetti principali di uno Stream Processing

Ci sono delle importanti caratteristiche e termini associati allo Stream Processing da premettere prima di analizzare le principali tecnologie esistenti attualmente:

- Garanzia di consegna: significa quale è la garanzia con cui, a prescindere da qualunque cosa accada, verrà elaborato un particolare record in arrivo al motore di Stream Processing. Possono esistere “at least once” (record processato almeno una volta anche in caso di fallimenti), “at most once” (record che può anche non essere processato in caso di fallimenti), “exactly once” (record processato una e una sola volta anche in caso di fallimenti). Ovviamente l’ultima semantica è la più desiderata, ma molto difficile da raggiungere nei sistemi distribuiti e comporta un tradeoff delle performance.
- Fault Tolerance: in caso di fallimenti, come per esempio nodi guasti, errori di rete e così via, la piattaforma deve essere in grado di ripristinare lo stato e ripartire a processare di nuovo da dove era rimasta in precedenza. Questo è spesso supportato tramite checkpoint dello stato dello Streaming in qualche supporto di storage persistente periodicamente nel tempo, per esempio checkpoint degli offset di Kafka a Zookeeper dopo aver preso un record da Kafka e averlo processato.
- Mantenimento dello stato: in caso di processamenti stateful, in cui dobbiamo mantenere stato, la piattaforma deve essere in grado di garantire meccanismi con cui preservare e aggiornare lo stato.
- Performance: fattore che include la latenza (quanto presto un record può essere processato), throughput (record processati al secondo) e scalabilità. La latenza deve essere minore possibile mentre lo throughput deve essere più alto possibile. La sfida è riuscire a garantire entrambi allo stesso tempo (molto difficile).
- Feature avanzate: Event Time Processing, Watermarks e Windowing sono funzionalità avanzate necessarie sotto particolari requisiti e complessità dello scenario applicativo.
- Maturità: è importante anche considerare l’adozione della piattaforma in questione; se è già stata provata e la scalabilità è già stata dimostrata può essere un’ottima soluzione.

### 1.5.2 Tipologie di Stream Processing

Esistono due approcci per implementare una piattaforma di Streaming:

- Native Streaming: ogni record in arrivo è processato appena possibile e il più presto possibile in relazione al suo arrivo, senza aspettare altri record. In questi scenari, ci sono dei processi perennemente attivi che processano i record che passano da loro. Esempi sono Storm, Flink, Kafka Streams, Samza.
- Micro-Batching: I record che arrivano in pochi secondi sono messi in batch insieme e processati in un singolo mini-batch con un ritardo di qualche secondo. Esempi sono Spark Streaming e Storm-Trident.

Native Streaming garantisce alla piattaforma di avere la minor latenza possibile, ma è difficile raggiungere il requisito di fault tolerance senza compromettere lo throughput, perché è necessario effettuare il checkpoint dei processamenti. Il mantenimento dello stato è semplice in quanto questi processi attivi possono mantenere lo stato richiesto facilmente.

Nel Micro-Batching si hanno caratteristiche opposte. Il fault tolerance viene di conseguenza, in quanto è essenzialmente un batch, e lo throughput è comunque alto nel processamento. Anche il checkpoint viene effettuato facilmente e velocemente, in quanto si utilizza un singolo checkpoint per ogni batch (gruppo di records), non per ogni record. Si perde però in latenza e nell'innaturalità dello Streaming. Il mantenimento dello stato è più difficile da gestire rispetto al Native Streaming.

## 1.5.3 Storm

Storm è una soluzione Apache per lo Stream Processing. È la più vecchia piattaforma di Streaming open source e una delle più mature e affidabili. È basato su un vero e proprio Streaming ed è molto utile in scenari basati su eventi.

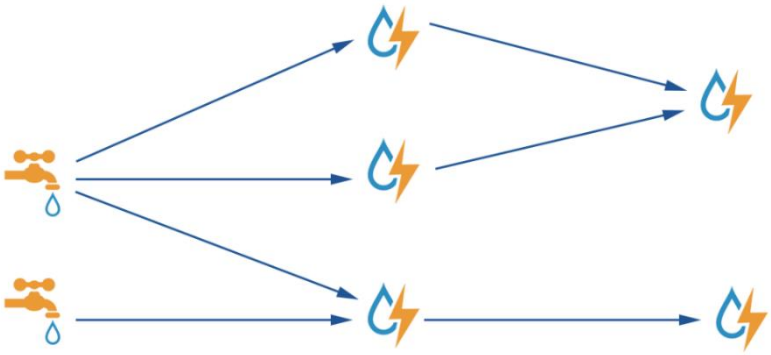

Figura 3. Apache Storm

Vantaggi principali di Storm:

- Latenza molto bassa.
- Vero Streaming.
- Maturo.
- Alto throughput.

Svantaggi:

- Non ha gestione dello stato.
- Non ha le feature avanzate.
- Ha solo “at least once” come semantica di garanzia.

## 1.5.4 Spark Streaming

Spark è una soluzione Apache nata come successore di hadoop in Batch Processing e come primo framework in grado di supportare a pieno la “Lambda Architecture” (in cui sia Batch che Streaming sono implementate, la prima per correttezza, la seconda per velocità). È molto popolare, maturo e ampiamente adottato. Da dopo la versione 2.0 è chiamato Streaming Strutturato ed è equipaggiato con tante buone funzionalità come una gestione custom della memoria, watermarks, event time processing e altro. Dalla versione 2.3 in poi c’è un’opzione che permette di switchare tra micro-batching e continuous streaming mode, il quale permette di avere molta meno latenza, come Storm e Flink.

Vantaggi:

- Lambda Architecture supportata.
- Alto throughput.
- Fault tolerance.
- Semplice da usare grazie alle API di alto livello.
- Grande community.
- Semantica “exactly once”.


Figura 4. Apache Spark

Svantaggi:

- Non Native Streaming
- Troppi parametri per effettuare tuning.
- Stateless per natura.

## 1.5.4 Flink

Flink ha un background simile a Spark. Supporta anch’esso la Lambda Architecture, ma l’implementazione è praticamente opposta a Spark. Spark è essenzialmente un batch con Spark Streaming come micro-batching, Flink è un Native Streaming, trattando il Batch come un caso speciale di streaming con dati limitati. Le API sono simili tra di loro, ma non hanno somiglianze nell’implementazione. In Flink ogni operazione di processamento è implementata come un operatore attivo perennemente.

Vantaggi:

- Leader di innovazione in Streaming open source.
- Primo vera piattaforma di Streaming (Native Streaming) con feature avanzate.
- Bassa latenza.
- Alto throughput.
- Auto-configurabile, con non troppi parametri per il tuning.
- Semantica “exactly once”.
- Ampiamente accettato da grosse e famose aziende.


Figura 5. Apache Flink

Svantaggi:

- Un po’ in ritardo nel mercato.
- Community non così grande come Spark.
- Non ci sono state adozioni del Flink Batch per adesso, è popolare solo per lo Streaming.

## 1.5.5 Kafka Streams

Kafka Streams, al contrario delle altre piattaforme di Stream Processing, è una libreria leggera molto utile e pratica per lo Stream Processing di dati da e verso Kafka. È una libreria molto simile a Java Executor Service Thread pool, ma con un supporto built-in per Kafka. È molto semplice da usare, da sviluppare e da mettere in esecuzione e, grazie alla sua natura leggera, può essere usata per microservizi. Internamente utilizza Kafka Consumer Group e lavora sul concetto di Kafka log. Ha la possibilità di abilitare la semantica “exactly once” tramite un flag, il quale la gestisce in modo automatico.


Figura 6. Apache Kafka

Vantaggi:

- Libreria veramente molto leggera (Microservizi/IoT).
- Non necessita di cluster dedicati.
- Eredita tutti i vantaggi di Kafka.
- Supporta join di Stream e utilizza RocksDb internamente per mantenere stato.
- Semantica “exactly once”.

Svantaggi:

- Accoppiata fortemente con Kafka.
- Inizia ad essere utilizzata solo di recente, non matura come altre possibili tecnologie.
- Non adatto a applicazioni con operazioni molto pese, come Spark Streaming e Flink.

### 1.5.6 Quale tecnologia scegliere?

Come spesso accade in ambiti ingegneristici e informatici, la risposta alla domanda “quale tecnologia è la migliore” è sempre “dipende”. In effetti, ogni tecnologia è nata per un motivo: cercare di essere ottimale in specifici contesti. Quindi, per capire quale tecnologia sia la migliore, bisogna contestualizzare l’ambito di utilizzo della stessa. Oltre questo, è necessario tenere a mente altre considerazioni, quali possibili utilizzi futuri dell’applicazione che stiamo sviluppando, la community della tecnologia, le tecnologie già presenti nell’applicazione fino ad ora ecc.

- Casi d’uso: se il contesto di utilizzo è semplice, non è necessario scegliere la tecnologia più complessa, avanzata e con più feature, perché sarebbe sia inutile, sia comporterebbe uno spreco di tempo, oltre che, probabilmente, un abbassamento del throughput. Spesso questa scelta dipende anche da quanto siamo disposti a spendere in relazione a ciò che vogliamo in output dall’applicazione. Per esempio, se la nostra applicazione gestisce dei semplici eventi, magari generati da device IoT, Storm o Kafka Streams sono perfette come tecnologie.
- Considerazioni future: è necessario, appunto, chiedersi quali saranno le possibili estensioni o utilizzi futuri dell’applicazione. Potrebbe rendersi necessario, in futuro, l’utilizzo di feature avanzate, quali Windowing o aggregazioni; in quel caso, è necessario propendere subito per tecnologie che offrono tali feature, in quanto renderebbe più estensibile e riutilizzabile l’applicazione. Ovviamente, potrebbe essere possibile implementare da zero le feature avanzate anche in tecnologie che non le offrono, ma comporterebbe uno sforzo sia a livello di tempo, sia a livello economico, e, oltretutto, la soluzione sviluppata sarebbe, con molte probabilità, non efficiente come quelle già implementate e testate da altri framework.
- Tecnologie già presenti nell’applicazione: è necessario tenere in considerazione le eventuali tecnologie già inglobate nell’applicazione; per esempio, se Kafka è già utilizzato come appoggio per altri scopi, probabilmente utilizzare Kafka Streams agevola lo sviluppo, in termini di compatibilità e di tempistiche. D’altra parte, se il processamento si basa su Lambda Architecture, potrebbe essere necessario considerare tecnologie come Spark Streaming o Flink Streaming.

# 2. Tecnologie

In questo capitolo verrà descritto lo stato dell'arte e le principali tecnologie utilizzate per lo sviluppo di questo progetto di tesi.

## 2.1 Stato dell'arte

Fino a pochi mesi fa, le soluzioni per effettuare statistiche di ogni tipo sui dati provenienti dai sensori in oggetto di questa tesi venivano messe in esecuzione con query su DB, in una modalità quindi sia "post mortem" che "by need", ma anche poco efficiente. Ovvero, nessuna tecnologia di Big Data era utilizzata per questi scopi. Nel momento in cui è diventata ingestibile la grande mole di dati di cui venivano fatte statistiche, sono state progettate delle soluzioni con nuove tecnologie.

I progetti attualmente sviluppati e in fase di sviluppo utilizzano la tecnologia basata su Kafka e Kafka Streams, con tutte le librerie e framework correlate e che verranno spiegate nel prossimo capitolo. Le statistiche che per adesso sono state implementate riguardano solo un conteggio di pacchetti (informazioni/transazioni) ricevute nel sistema dagli OBU, sia differenziate per ogni OBU che singolarmente.

Sono presenti dei moduli che raggruppano per finestre temporali i pacchetti ricevuti ed effettuano il conteggio su queste finestre. Per adesso sono state sviluppate finestre temporali di quindici minuti e di trenta minuti, di cui una da quindici minuti "generale", ovvero senza raggruppamento per OBU, mentre altre due, una da quindici e una da trenta, raggruppate per OBU. In figura 7 è descritto il workflow generale del processamento dei dati provenienti dai tratti autostradali.

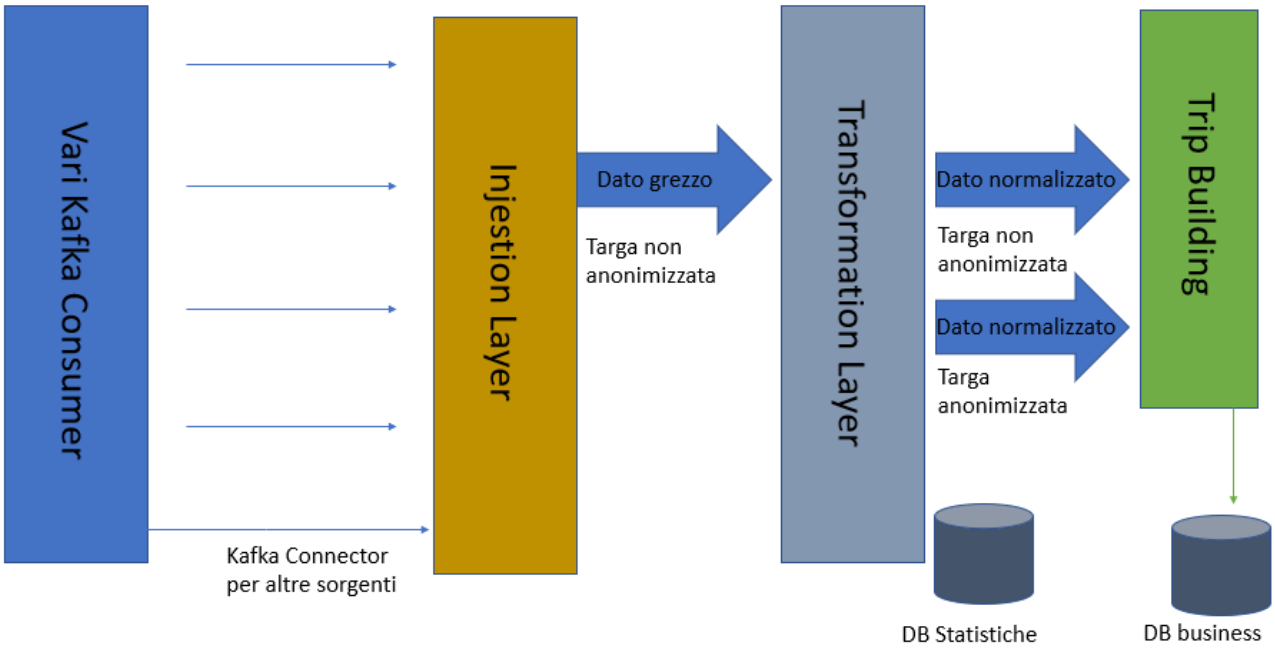


Figura 7. Workflow processamento dati

Ovviamente, i dati che arrivano dai sensori potrebbero non essere anonimizzati, per esempio la lettura di una targa: è quasi sempre necessario effettuare un preprocessamento prima di arrivare alle operazioni di analytics, in quanto nelle statistiche i dati non devono, generalmente, avere nominativi, soprattutto per questioni di privacy. È possibile fare un'ulteriore considerazione riguardante l'anonimizzazione dei dati: ovviamente, per sistemi di pedaggio o di tutor (multa in caso di superamento puntuale del limite di velocità), il diretto interessato deve essere notificato affinchè possa effettuare il pagamento; ecco, perché, non tutti i dati sono effettivamente anonimizzati, in quanto quelli strettamente necessari hanno un nominativo associato.

Il progetto attuale consta anche di un modulo che espone delle API semplici ed efficienti per poter accedere alle statistiche. Questo modulo è sviluppato tramite REST API con Spring, che verrà brevemente introdotto e illustrato nel prossimo capitolo.

La piattaforma con cui sono state ampliate le funzionalità preesistenti è, come già accennato, Apache Kafka. Oltre a questa, per l'ambiente di test è stato necessario l'utilizzo di Docker e Docker Compose. Infine, per rendere accessibili in maniera semplice le statistiche, è stata utilizzata la tecnologia Spring tramite Spring Boot, per le REST API.

## 2.2 Apache Kafka

Apache Kafka è una piattaforma open-source per data streaming distribuito sviluppata inizialmente da Linkedin, utilizzato principalmente per data pipeline ad alta performance, streaming analytics e data integration. È progettata per gestire enormi flussi di dati provenienti da uno o più produttori, distribuendoli a uno o più consumatori.

Nel mondo odierno, la forte propensione per microservizi e la riduzione delle dipendenze hanno accelerato lo sviluppo di applicazioni distribuite: quest'ultime devono comunque integrarsi per consentire la condivisione dei dati. Esistono due modalità per effettuare tale condivisione: sincrona, in cui si sfruttano interfacce API; asincrona, in cui si ha una replica dei dati in archivi intermedi. Apache Kafka sfrutta l'integrazione asincrona, infatti i dati provenienti da, per esempio, altri team di sviluppo vengono trasmessi nell'archivio dati intermedio e successivamente condivisi con altri team e applicazioni. Affinché possa funzionare correttamente questo metodo di integrazione, è necessario che siano presenti tre proprietà fondamentali:

1. Integrazioni distribuite: integrazioni leggere basate sulla distribuzione nei nodi che ne hanno bisogno, senza avere le limitazioni di ESB centralizzati.
2. API: riescono a portare il massimo beneficio dai servizi verso clienti e consumatori.
3. Container: per riuscire a sviluppare, gestire e far scalare applicazioni. I container riescono a rendere i componenti agili, distribuibili e altamente disponibili.

L'integrazione diventa una fase dello sviluppo applicativo, portando benefici quali agilità e adattabilità.
Apache Kafka viene utilizzato in contesti in cui l'integrazione tra sistemi o applicazioni ha bisogno di una condivisione dati rapida, efficiente e scalabile. Si ha una latenza dell'ordine di millisecondi grazie alla riduzione di integrazioni P2P e alle caratteristiche intrinseche della piattaforma, che la rendono tra le migliori del suo genere.

Le problematiche in cui si inserisce Apache Kafka sono molteplici, ma tra le tante spiccano quelle dei Big Data e IoT.

## 2.2.1 Caratteristiche funzionali

Apache Kafka, come già ribadito, ha lo scopo di ottimizzare la trasmissione ed elaborazione di flussi di dati che vengono scambiati tramite collegamento diretto tra destinatario e fonte di dati. Il suo ruolo diventa come quello di un'istanza di messaggistica tra mittente e destinatario, offrendo soluzioni alle tipiche difficoltà di questo contesto. Kafka sviluppa una coda di messaggi, che sfrutta per implementare le soluzioni a tali problematiche. Una coda ben configurata impedisce il sovraccarico del destinatario da parte del mittente, per esempio. Inoltre, se il destinatario collassa durante l'elaborazione del messaggio, il mittente è notificato con un errore. Quindi, si ha tolleranza ai guasti, alta scalabilità, forte capacità di distribuzione e alta disponibilità, che rendono Apache Kafka molto appetibile nei contesti citati nel paragrafo precedente.

Apache Kafka viene eseguito su un cluster di nodi, tipicamente su uno o più server. Ogni nodo di un cluster viene chiamato broker e ha il compito di salvare e categorizzare il flusso di dati in entrata nei topic. Vengono poi suddivisi in partizioni, replicati e distribuiti nel cluster, per poi essere contrassegnati con una marca temporale. Si ha una differenziazione del topic in due categorie: normal topic, in cui Kafka può cancellare i messaggi una volta superato il periodo o il limite di salvataggio; compacted topic, i cui dati che contiene non hanno limite né spaziale né temporale.

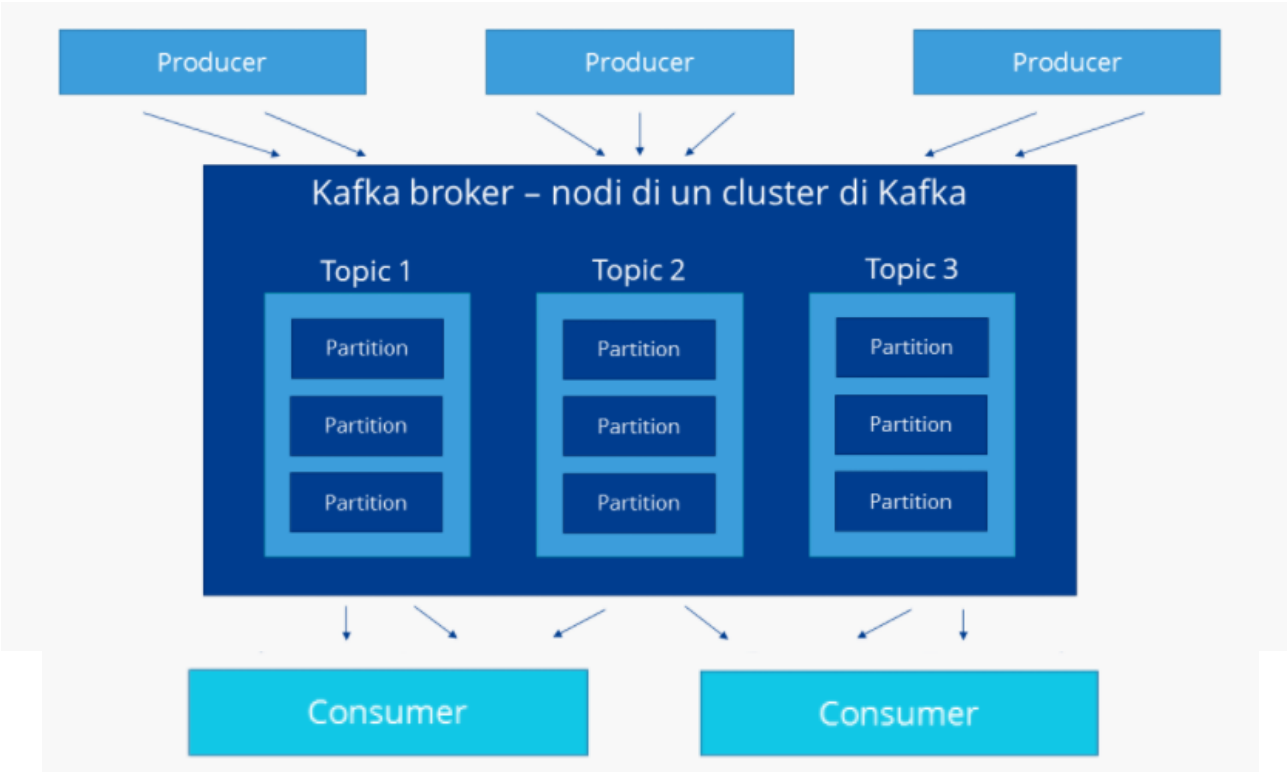


Figura 8. Overview distribuzione Kafka

Le applicazioni che scrivono dati su un cluster di Kafka vengono definiti produttori (producers), mentre quelle che leggono i dati sono chiamati consumatori (consumers). La componente centrale da cui attingono produttori e consumatori è una biblioteca Java chiamata Kafka Streams (in realtà, non è l'unica configurazione per poter utilizzare Kafka, ma è fortemente consigliata). Si hanno cinque interfacce principali:

- Kafka Producer: interfaccia che permette alle applicazioni di inviare i flussi di dati ai broker di un cluster di Kakfa per categorizzarli e salvarli.
- Kafka Consumer: i consumatori possono ricevere accesso ai dati salvati nei topic tramite questa interfaccia.
- Kafka Streams: permette a un'applicazione di attivarsi come processore di streaming, per trasformare i flussi di dati in ingresso in dati in uscita.
- Kafka Connect: è possibile impostare produttori e consumatori riutilizzabili che collegano i topic Kafka con le applicazioni o banche dati esistenti.
- Kafka AdminClient: offre un'amministrazione e ispezione di un cluster Kafka.

La comunicazione tra applicazioni del cliente e singoli server di un cluster Kafka avviene tramite un protocollo indipendente dal linguaggio sulla base di un TPU. Solitamente vengono utilizzati cliente Java, ma esistono anche altri linguaggi che si possono utilizzare: PHP, Python, C/C++, Ruby, Perl, Go.

### 2.2.1.1 Kafka Broker

Un Kafka cluster consiste in uno o più server, chiamati Kafka Broker, che eseguono Kafka. Sfruttare un singolo Kafka Broker per un contesto applicativo è possibile, ma non dà modo di sfruttare tutti i benefici di Kafka, per esempio data replication.

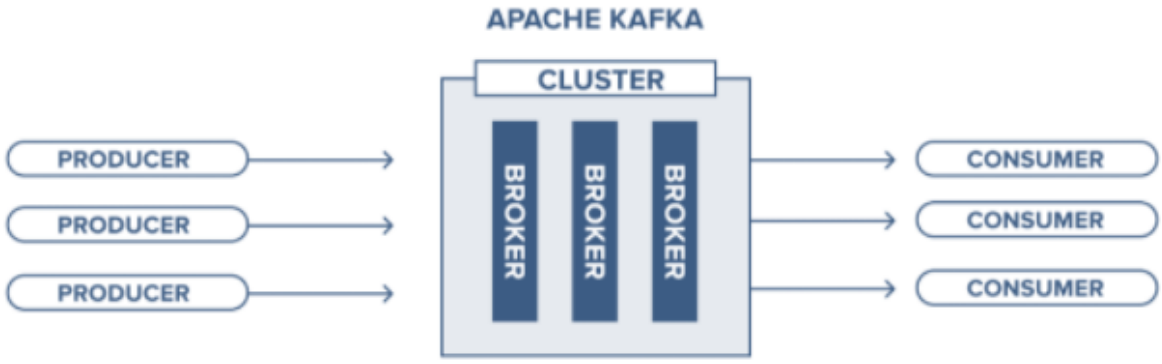


Figura 9. Schema interazione

La gestione dei broker in un cluster è fatta da Zookeeper. Possono esserci molteplici istanze di Zookeeper in un cluster: la raccomandazione è di avere da 3 a 5 istanze attive, cercando di tenerne in un numero dispari, così che ci sia sempre una maggioranza, ma anche in numero minore possibile, per prevenire un overhead di risorse.

### 2.2.1.2 Kafka Topic

Un topic è un nome di categoria in cui i dati sono memorizzati e pubblicati dai produttori. Tutti i record Kafka sono organizzati in topic, infatti i produttori scrivono data sui topic, i consumatori li leggono. Record pubblicati in un cluster rimangono nel cluster finché un “retention period” non è trascorso. Kafka conserva i record in dei file (log), rendendo i consumatori responsabili del monitoraggio della posizione nel registro, noto come "offset". In genere, un consumatore avanza l'offset in modo lineare man mano che i messaggi vengono letti. Tuttavia, la posizione è effettivamente controllata dal consumatore, che può consumare i messaggi in qualsiasi ordine. Ad esempio, un consumatore può reimpostare un offset precedente durante la rielaborazione dei record.

### 2.2.1.3 Log

È stato accennato che i record dei topic sono memorizzati in dei file chiamati log. I log sono un elemento cruciale per la gestione dei topic in Kafka. I log sono dei file visti come “append-only” e come insieme di record totalmente ordinati nel tempo, ma non tramite dei timestamp, ma grazie alla loro posizione nel file stesso. Infatti, il primo record che arriva viene messo a sinistra, gli altri tutti appesi a destra fino all’ultimo che sta nell’ultima posizione a destra. Possiamo quindi dire che i topic in Kafka sono dei file log gestiti tramite nomi.

Durante il setup di Kafka, è necessario specificare la directory in cui i file di log saranno memorizzati. Ogni topic si mappa in una sottodirectory sotto la directory specificata di log. Possono esserci diverse sottodirectory a seconda di quante partizioni ci sono (saranno affrontate nel prossimo paragrafo), con un formato del tipo “nomepartizione_numeropartizione”. Una volta che un file raggiunge una certa dimensione prefissata oppure passa un certo intervallo di tempo, il file è sostituito da uno nuovo.

Ci possono essere due tipologie di approcci a seguito del raggiungimento dei prerequisiti per sostituire il file di log:

- Eliminazione dei log: è un approccio a due fasi. Prima si raggruppano i log in segmenti, successivamente il segmento più vecchio viene eliminato. Per effettuare il raggruppamento,

Kafka utilizza timestamp inglobati nei messaggi. Kafka separa i log nel caso in cui il timestamp di un nuovo messaggio in arrivo è maggiore del timestamp del primo messaggio nel log più log.roll.ms, valore configurabile. Al tempo stesso, esistono le policy per quando eliminare i log: log.retention.ms (tempo massimo per tenere il file di log) e log.retention.bytes (dimensione massima dei file di log).

- Compattazione dei log: immaginiamo di avere dati con chiave e stiamo ricevendo aggiornamenti di questi dati nel tempo, ovvero record con la stessa chiave saranno aggiornati. Al posto di utilizzare un approccio coarse-grained, in cui eliminiamo interi segmenti basandoci sul tempo o sullo spazio che occupano in memoria, si sfrutta un approccio fine-grained, in cui eliminiamo i record raggruppati per chiave. Ovvero, vengono lanciati dei thread in background che si occupano di ricopiare i segmenti dei log e rimuovere record se c'è un'occorrenza successiva con la stessa chiave in quel log.

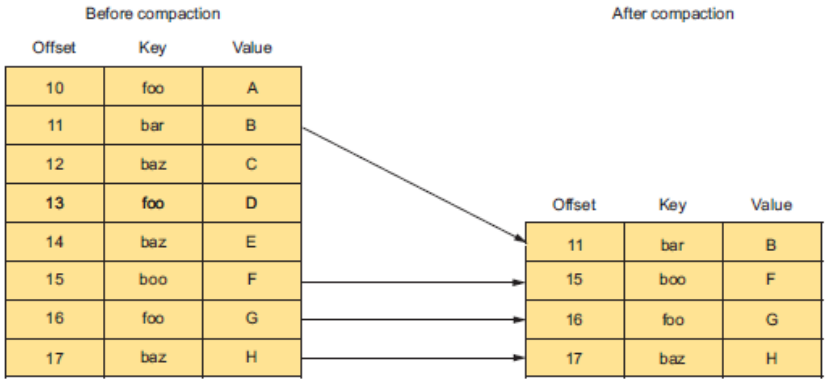


Figura 10. Esempio compattazione log

## 2.2.1.4 Kafka topic partition

I topic sono suddivisi in un certo numero di partizioni, le quali contengono record in una sequenza non modificabile. Ogni record in una partizione è identificato da un unico offset. Un topic può anche avere più registri di partizione, che consente a molteplici consumatori di leggere da un topic in parallelo. Le partizioni consentono ai topic di essere parallelizzati dividendo i dati in un topic particolare tra i molteplici broker.

La replicazione in Kafka è implementata al livello di partizione. Ogni partizione ha generalmente una o più repliche, ciascuna delle quali è un'unità ridondante di una topic partition; quindi, le partizioni contengono messaggi che sono replicati in alcuni Kafka broker nel cluster. Ogni partizione (replica) ha un server che si comporta come leader e gli altri con follower. Il leader gestisce ogni richiesta di

scrittura e lettura da parte di una specifica partizione, mentre i follower replicano le azioni del leader. Sarebbe opportuno di bilanciare correttamente i leader e i follower, in modo che ciascun broker sia un leader di un egual numero di partizioni, per distribuire il carico. Se il leader fallisce, uno dei follower diventerà il nuovo leader. Quando un produttore pubblica un record in un topic, lo fa attraverso il suo leader. Il leader inserisce il record nel suo registro e incrementa il record offset.

Kafka espone un record a un consumatore solo dopo che è stato fatto il commit e ogni pezzo di dato che arriva è stato inserito nel cluster. Un produttore deve sapere in quale partizione scrivere, non è compito del broker indirizzarlo. È possibile da parte del produttore di inserire una chiave nel record che definisce in quale partizione quel record dovrà essere trasmesso. Tutti i record con la stessa chiave arriveranno nella stessa partizione. Prima che un produttore possa mandare un record, devono essere richiesti dei metadati riguardo il cluster al broker. Ogni metadato contiene informazioni su quale broker è il leader per ogni partizione e un produttore scriverà sempre al partition leader.

#### 2.2.1.5 Consumatori

Ci sono due tipi di consumatori:

- Low-Level consumers: topic e partizione sono specificati come l'offset da cui leggere, entrambi in posizione fisso (all'inizio o alla fine). Questo può essere poco maneggevole ai fini di tenere traccia di quali offset sono stati consumati, per prevenire la doppia lettura di record. Per questo è stato aggiunto un altro tipo di consumatori.
- High-Level consumers: conosciuti anche come consumer groups, consistono in uno o più consumatori. Il consumer group è creato aggiungendo una proprietà "group.id" a un consumatore. Consumatori con lo stesso id fanno parte dello stesso gruppo. Il broker distribuirà in base a quale consumatore deve leggere da quali partizioni e tiene traccia anche dell'offset in cui si trova il gruppo per ciascuna partizione. Ogni volta che un consumatore è aggiunto o rimosso da un gruppo, la consumazione è ribilanciata nel gruppo. Tutti i consumatori sono fermati durante il ribilanciamento, così i clienti che "scadono" o vengono riavviati spesso ridurranno la velocità effettiva. È importante rendere i consumatori stateless, in quanto dopo un ribilanciamento potrebbe essere associato a partizioni diverse. I consumatori prendono messaggi dai topic partitions. Consumatori differenti possono essere responsabili di differenti partizioni. Kafka può supportare un grande numero di consumatori e conservare grandi quantità di dati con un piccolo overhead. Usando i consumer groups, è possibile parallelizzare i consumatori, come citato sopra, quindi si ha un alto throughput di

processamento di messaggi. I record non saranno mai pushati ai consumatori autonomamente, ma saranno inviati quando i consumatori li richiedono e sono pronti per gestirli.

### 2.2.1.6 Record flow

Prendiamo in considerazione un comportamento generale di Kafka e dei record. Abbiamo come esempio un broker con tre topic, ogni topic ha otto partizioni.

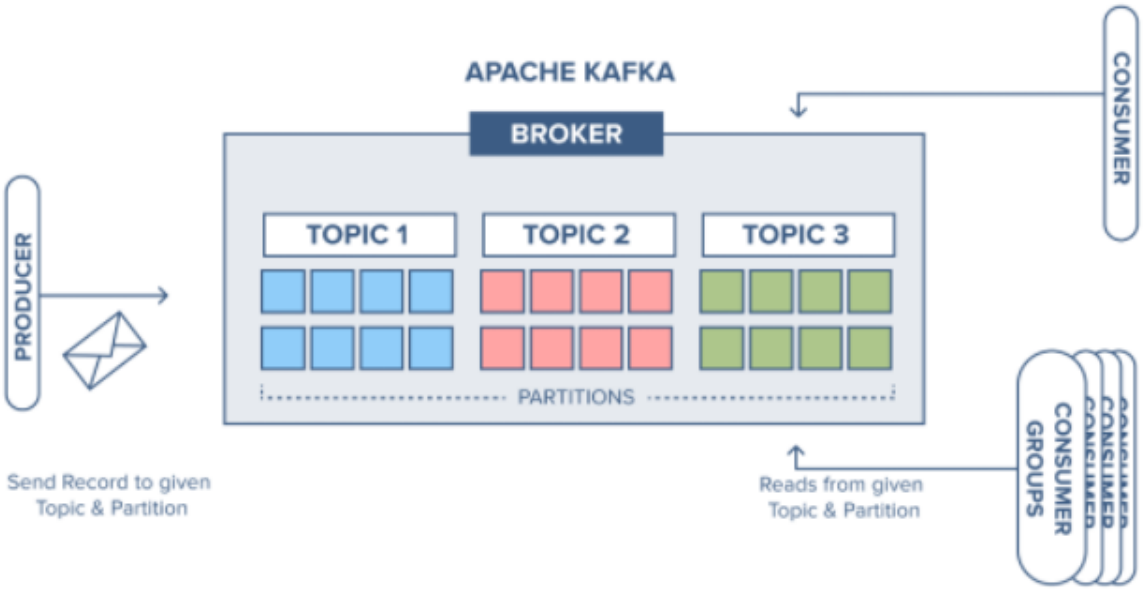


Figura 11. Esempio interazione e workflow 1

Il produttore manda un record alla partizione 1 del topic 1 e, poiché la partizione è vuota, il record avrà offset 0.

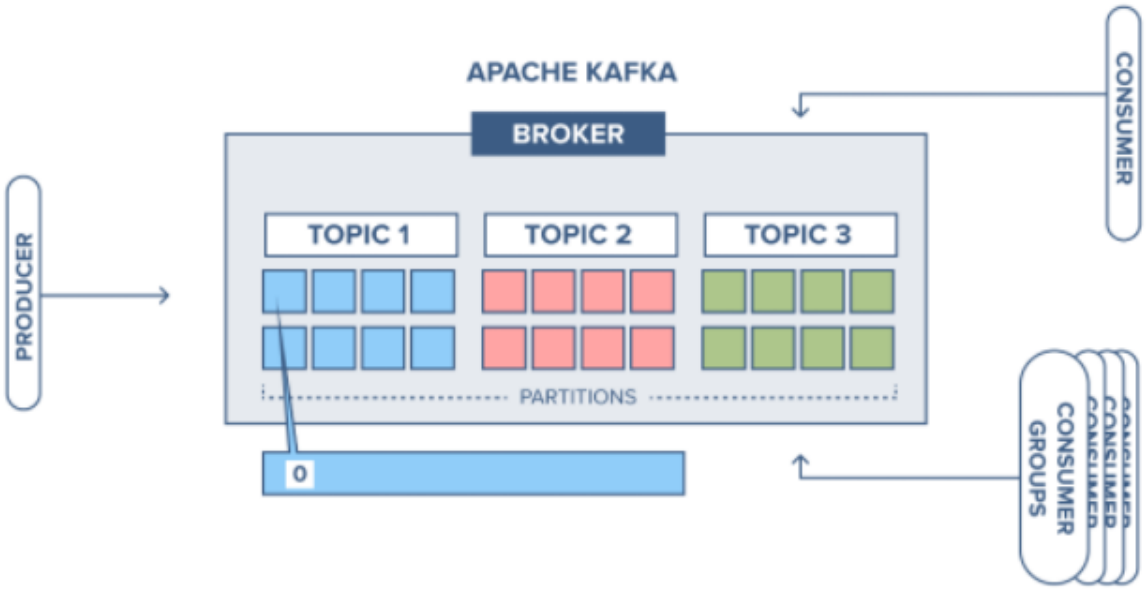


Figura 12. Esempio interazione e workflow 2

Continuando a mandare messaggi, avremo che nel topic 1 si accumuleranno i record con offset crescenti, 1, 2, 3, ecc.

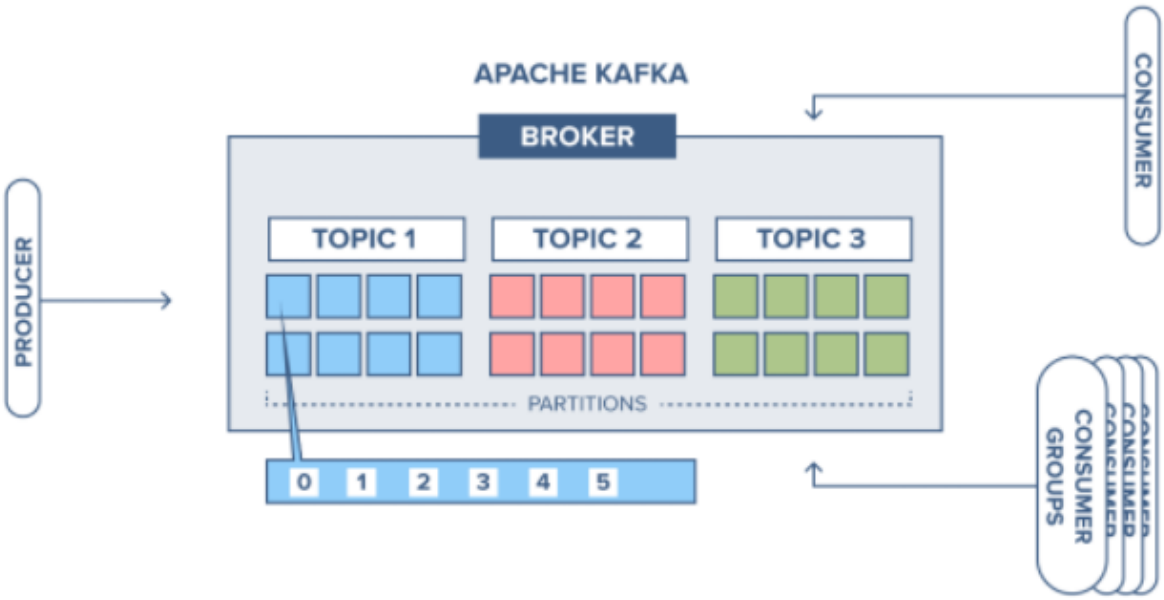


Figura 13. Esempio interazione e workflow 3

Ogni record è riferito a un commit log, senza possibilità di modifica.

## 2.2.2 Kafka Consumer

Analizziamo nel dettaglio cosa sono i Kafka Consumer e quali sono le principali caratteristiche. Per cominciare, si parlerà di Consumer Groups.

### 2.2.2.1 Kafka Consumer Groups

Un consumer group è un insieme di consumer che cooperano per consumare dati da alcuni topic. Le partizioni di ogni topic sono divise tra i consumer di un gruppo. Quando un nuovo membro arriva e/o quando un membro vecchio se ne va, le partizioni sono riassegnate in modo tale da creare una condivisione proporzionale delle partizioni tra i membri attuali del gruppo (ribilanciamento del gruppo). Il nuovo protocollo di gestione dei gruppi è inglobato in Kafka stesso. Uno dei broker è designato come coordinatore del gruppo ed è responsabile della gestione dei membri del gruppo stesso, insieme all'assegnamento delle loro partizioni.

Il coordinatore di ogni gruppo è scelto dai leader di una topic interna, __consumer_offsets, che è usata per memorizzare gli offset dopo il commit. Generalmente, viene fatto l'hash dell'id del gruppo su una delle partizioni di un topic e il leader di quella partizione viene selezionato come coordinatore. Così, la gestione dei gruppi di consumatori è divisa circa equamente tra tutti i broker del cluster, cosa che consente di scalare discretamente bene.

Quando un consumatore si attiva, trova il coordinatore per il suo gruppo e gli manda la richiesta di join per quel gruppo stesso. Il coordinatore fa, dunque, partire il ribilanciamento del gruppo, in modo tale da assegnare le partizioni correttamente. Ogni ribilanciamento dà come output una nuova formazione del gruppo. Ogni membro del gruppo deve mandare heartbeat al coordinatore, in modo tale da rimanere un membro attivo all'interno del gruppo. Se non è ricevuto un hearbeat dopo un certo session timeout preconfigurato, allora il coordinatore espelle il membro dal gruppo e riapplica il ribilanciamento.

#### 2.2.2.2 Gestione degli offset

Dopo che il consumatore ha ricevuto il suo assegnamento dal coordinatore, deve determinare la sua posizione iniziale per ogni partizione assegnata. Quando il gruppo è creato per la prima volta, prima che un qualunque messaggio venga consumato, la posizione è impostata seguendo una policy configurabile, chiamata "auto.offset.reset". Tipicamente, il consumo parte o dal più vicino offset o dal più lontano (earliest/latest).

Quando un consumatore di un gruppo legge messaggi dalle partizioni assegnate dal coordinatore, deve fare il commit degli offset corrispondenti ai messaggi che ha letto. Se il consumatore va in crash o si spegne, le sue partizioni sono riassegnate ad un altro membro, che inizierà il consumo dall'ultimo offset di cui è stato fatto il commit per ogni partizione. Se il consumatore va in crash prima di aver fatto un qualunque commit, allora il nuovo consumatore si adeguerà alla policy di reset configurata.

La policy di commit degli offset è fondamentale a garantire la consegna del messaggio. A default, il consumatore è configurato per usare una policy di commit automatico, che triggera un commit a intervalli periodici, oltre a supportare una API di commit che può essere usata per effettuare il managing degli offset in manuale.

Usando l'auto-commit, Kafka si impegna a garantire la semantica "at least once", ossia nessun messaggio sarà perduto, al massimo ci possono essere duplicazioni. L'auto-commit può essere impostato da "auto.commit.interval.ms", proprietà configurabile. Se il consumatore va in crash, dopo un restart o un ribilanciamento la posizione delle partizioni del consumer crashato sono resettate all'ultimo offset di cui è stato fatto il commit. Quando questo accade, quest'ultimo può essere più vecchio dell'intervallo di auto-commit: ogni messaggio arrivato dall'ultimo commit sarà letto di nuovo.

Se vogliamo ridurre la finestra di duplicazione, è necessario ridurre l'intervallo di auto-commit, ma spesso è utile avere sotto controllo i commit, quindi è possibile utilizzare le apposite API. Se volessimo fare così, è necessario disabilitare l'auto-commit mettendo la proprietà "enable.auto.commit" a "false". Se utilizziamo le API di commit sincrone, il consumatore è bloccato quando chiede al broker di effettuare il commit, quindi riduciamo lo throughput, in quanto il consumatore stesso, mentre aspetta la risposta dal broker, avrebbe potuto consumare altri messaggi. Una possibile strada è di aumentare il numero di informazioni che viene restituito quando il consumatore effettua il polling. Per fare questo, è possibile utilizzare le proprietà "fetch.min.bytes", che controlla quanti dati devono essere restituiti per ogni fetch, e "fetch.max.wait.ms", che è il tempo

massimo di attesa nel caso in cui nessun dato sia disponibile. Questo può però aumentare il numero di duplicati nel caso ci siano crash o guasti.

Una seconda opzione è quella di utilizzare i commit asincroni. Invece di aspettare la risposta dal broker, il consumatore può tornare immediatamente al controllo, senza attesa. Questo incrementa sicuramente le performance, ma ha anche degli svantaggi. Se la richiesta di commit fallisce, non riusciamo a farla riprovare al consumer, in quanto non attendiamo la risposta dal broker. Questa cosa, invece, ci veniva gratuita con le API sincrone. Inoltre, si ha il problema dell'ordinamento dei commit. Se il consumer si accorge che un commit è fallito, potrebbe già aver processato il prossimo batch di messaggi e aver mandato tale commit. In questo caso, una nuova richiesta di commit del vecchio batch può causare un consumo duplicato.

Per cercare di non complicare l'implementazione del consumer con tutte le prove di commit all'interno di esso, le API offrono una funzione di callback che viene invocata quando il commit ha successo o fallisce. È possibile utilizzarla per ritrasmettere il commit in caso di fallimento, anche se presenta gli stessi problemi di ordinamento.

Un pattern comune per gestire il commit degli offset è quello di utilizzare commit asincroni nel caso ci troviamo nel pool loop di un consumer e commit sincroni nel caso di ribilanciamento o crash.

Ogni ribilanciamento ha due fasi: revoca della partizione e assegnazione della partizione. La revoca è chiamata sempre prima del ribilanciamento stesso ed è l'ultima chance per un commit degli offset prima che le partizioni siano riassegnate. L'assegnazione è chiamata dopo il ribilanciamento e può essere usata per impostare la posizione iniziale delle partizioni assegnate.

In generale, si considera meno safe il commit asincrono rispetto a quello sincrono: infatti, fallimenti consecutivi di commit prima di un crash aumenta notevolmente i duplicati. Potremmo mitigare questo problema aggiungendo gestione di controllo nella funzione di callback, ma è meglio non esagerare e non aumentare troppo la complessità. Se c'è bisogno di molta affidabilità, è necessario passare al commit sincrono. Se invece abbiamo bisogno di performance, senza problemi nel caso di un alto numero di duplicati, va bene il commit asincrono.

Per una semantica "at least once", il commit asincrono ha senso. Se volessimo una semantica "at most once", sarebbe necessario sapere, prima di consumare i messaggi, se il commit è andato a buon fine o no. Per fare questo bisogna utilizzare il commit sincrono.

### 2.2.3 Kafka Connect

Kafka Connect è uno strumento per lo streaming dati affidabile e scalabile tra Apache Kafka e altri data systems. Offre la possibile di definire facilmente dei connector che trasmettono grandi quantità di dati dentro e fuori Kafka. Kafka Connect può fare ingestion di interi database o collezionare dati da application server dentro Kafka topic. Un connector di export può mandare i dati da Kafka topic a sistemi secondari, come Elasticsearch o in batch systems come Hadoop, per analisi offline.

Kafka Connect lavora come un data hub centralizzato per una semplice integrazione di dati tra database, search indexes, file systems ecc.

Kafka Connect ha tre principali benefici:

- Pipeline incentrata sui dati: Connect usa astrazioni di dati significativi per fare pull o push a Kafka.
- Flessibilità e scalabilità: Connect può operare come streaming e batch system sia in un singolo nodo (standalone) che nel distribuito.
- Riusabilità e estensibilità: Connect riutilizza connettori esistenti o li estende per gestire le necessità per ogni contesto, garantendo basso tempo di produzione.

Kafka Connect è uno strumento che integra in una pipeline ETL, quando combinato con Kafka e un framework di stream processing.

Kafka Connect garantisce una bassa barriera di utilizzo e poco overhead di computazione. È possibile partire in piccolo con un deploy standalone, per poi incrementare e scalare in un ambiente di piena produzione, che supporta una grande quantità di dati e pipeline.

Esistono due tipi di connettori:

- Source Connector: Fa l'ingestion di interi database e stream in Kafka topic. Può anche collezionare dati da tutti gli application server e memorizzarli in Kafka topic, per rendere tali dati disponibile per il processamento a bassa latenza.
- Sink Connector: Trasmette dati da Kafka topic a sistemi esterni/secondari.

Le principali caratteristiche di Kafka Connect sono:

- Framework per connettere sistemi esterni con Kafka: semplifica lo sviluppo, il deployment e la gestione dei connector.

- Modalità distribuita e standalone: aiuta a effettuare il deploy su grandi cluster, oltre a sviluppo, test e deployment in piccole produzioni.
- Interfaccia REST: possibilità di gestire i connetor utilizzando le REST API.
- Integrazione di streaming e batch.

### 2.2.3.1 Funzionamento e connector

Per capire il funzionamento generale, prendiamo un esempio semplice: file source connector e file sink connector. È necessario, prima di tutto, configurare il file source connector tramite una serie di proprietà:

- name: nome per l'istanza del connector, specificata dall'utente.
- connector.class: specifica la classe di implementazione del source connector, ovvero il tipo di connector
- tasks.max: specifica quante istanze di un source connector devono eseguire in parallelo
- topic: definisce il topic al quale il connector deve mandare l'output.
- file: proprietà specifica del tipo di connector che abbiamo adesso (file source) che definisce il file da cui il connector legge in input.

Dopo di che, si configura il sink connector con praticamente le stesse proprietà sopra elencate. IN questo caso, il connector.class identifica la classe del sink connector e file identifica dove il connector dovrà scrivere il contenuto.

Successivamente, è necessario configurare il connect worker, il quale integrerà i due connettori, leggendo dal source connector e scrivendo al sink connector. Le principali proprietà da configurare sono:

- bootstrap.servers: contiene gli indirizzi dei Kafka broker.
- key.converter e value.converter: definiscono le classi di conversione, che serializzano e deserializzano i dati mentre fluiscono dal source connector dentro Kafka e poi da Kafka al sink connector.
- key.converter.schemas.enable e value.converter.schemas.enable: sono proprietà specifiche per la conversione.
- offset.storage.file.filename: proprietà più importante quando Connect è in modalità standalone, definisce infatti dove Connect deve memorizzare i suoi offset data.

- offset.flush.interval.ms: definisce l'intervallo con il quale il worker prova a fare commit degli offset.

Un'altra possibile configurazione di Connect che possiamo sfruttare è quella basata su REST API. A default, si hanno diversi endpoint a cui possiamo accedere:

- GET /connectors: restituisce una lista dei connettori in uso.
- GET /connectors/{name}: restituisce i dettagli di uno specifico connector.
- POST /connectors: crea un nuovo connector, il body deve essere un oggetto JSON contenente un campo stringa "name" e un campo oggetto "config" con le configurazioni del connector.
- GET /connectors/{name}/status: restituisce lo stato corrente del connector.
- DELETE /connectors/{name}: rimuove un connector, fermando i task e eliminando le sue configurazioni.
- GET /connector-plugins: restituisce una lista di connector istallati nel cluster Kafka Connect.

Esempio per creare un connector tramite REST API: immaginiamo di avere Kafka Connect in esecuzione in localhost/8083. Per creare un connector è necessario mandare due richieste POST a localhost/8083/connectors, includendo due file JSON, uno per ogni richiesta, con le configurazioni dovute.

```
{
    "name": "file-source",
    "config": {
        "connector.class": "FileStreamSource",
        "tasks.max": 1,
        "file": "test.txt",
        "topic": "my_topic"
}
```

```
{
    "name": "file_sink",
    "config": {
        "connector.class": "FileStreamSink",
        "tasks.max": 1,
        "file": "test_sink.txt",
        "topics": "my_topic"
}
```

Figure 14 e 15. Esempi JSON configurazione connettori

### 2.2.3.2 Trasformazioni

Le trasformazioni consentono di effettuare in maniera semplice ed efficiente modifiche a messaggi individuali. Kafka Connect supporta diverse trasformazioni predefinite:

- InsertField: aggiunge un campo utilizzando sia dati che metadati.
- ReplaceField: Filtra o rinomina campi.
- MaskField: sostituisce un campo con il valore "nullo" valido per il tipo corrispondente (per esempio, 0 per un intero, stringa vuota per una stringa ecc).
- HoistField: incapsula l'evento in un singolo campo all'interno di una mappa/struttura.
- ExtractField: estrae uno specifico campo dalla mappa/struttura e include solo questo campo nei risultati.
- SetSchemaMetadata: modifica il nome dello schema o la versione.
- TimestampRouter: modifica il topic di un record basandosi sul topic originale e il timestamp.
- RegexRouter: modifica il topic di un record basandosi sul topic originale, una stringa e una regular expression.

Una trasformazione può essere configurata tramite apposite proprietà:

- transforms: un insieme di alias di trasformazioni, separati da virgola.
- transforms.$alias.type: nome della classe per la trasformazione.
- transforms.$alias.$transformationSpecificConfig: configurazione per la trasformazione specificata.

### 2.2.3.3 Connector Custom

Per poter implementare un connector custom utilizzando Java, i passi da fare sono semplici. Innanzitutto, è necessario estendere la classe astratta SourceConnector e implementare tutti i metodi presenti, per poter creare il source connector. Alcuni metodi sono:

- start: uno dei primi metodi chiamati all'inizializzazione del connector. Qua dovremmo impostare degli stati interni per memorizzare delle proprietà che sono state passate da Kafka Connect. Importante riuscire a gestire eventuali errori nelle proprietà, per esempio porprietà obbligatorie non passate o proprietà non conformi a altri tipi di validazioni.
- taskClass: indichiamo qua quale connector task dovrebbe essere linkato al nostro connector.
- taskConfigs: metodo in cui si definisce la configurazione che ogni task dovrebbe avere. Gvali

- validate: valida i valori di configurazione del connector a fronte delle definizioni di configurazione.
- stop: metodo chiamato allo spegnimento del Connect Service o quando l'utente decide di non volere più il connector attivo.
- config: specifichiamo quali sono le proprietà di cui il connector deve tenere conto.

### 2.2.4 Avro

Sebbene possa sembrare una parentesi azzardata e non faccia parte di Apache Kafka, parlare di Avro ora è necessario, oltre che utile, per motivi che saranno ovvi a breve. Avro è un sistema per serializzazione di dati indipendente dal linguaggio di programmazione e basato su schema. Utilizza JSON per dichiarare le strutture dati, in modo da rendere più facilmente integrabile il tutto con linguaggi che abbiano già librerie per il supporto a JSON. Principali caratteristiche di Avro:

- Indipendente dal linguaggio di programmazione.
- Supportato correntemente da C, C++, C#, Python, Java e Ruby.
- Crea un formato strutturato binario che è sia comprimibile che divisibile. Può quindi essere utilizzato in modo efficiente come imput per MapReduce di Hadoop.
- Offre un ricco insieme di strutture dati.
- Schema definiti in JSON.
- Crea un file auto-descrittivo.
- È usato anche in RPC.

Principali passi da fare per utilizzare Avro:

- Creare gli schema, a seconda dei dati che dobbiamo gestire
- Leggere gli schema dentro al nostro programma, o generando una classe che corrisponde allo schema o utilizzando dei parser.
- Serializzare i dati utilizzando le API fornite da Avro (org.apache.avro.specific)
- Deserializzare i dati utilizzando le API di Avro.

### 2.2.4.1 Utilizzo di Avro

Avro segue degli standard per definire gli schema:

- tipo: tipo di file, record a default.
- posizione: dove si trova il record
- nome: del record
- campi: campi del record con i corrispondenti tipi di dato.

Analizziamo meglio come definire uno schema e come utilizzarli.

```
{
    "type" : "record",
    "namespace" : "prova",
    "name" : "Persona",
    "fields" : [
        { "name" : "Nome" , "type" : "string" },
        { "name" : "Età" , "type" : "int" }
    ]
}
```

Figura 16. Esempio schema Avro

- type: questo campo appare sia fuori che dentro i vari campi. Quando è fuori, identifica il tipo di documento, generalmente un record in quanto si hanno molteplici campi; quando dentro un campo, descrive il tipo di dato.
- namespace: descrive il nome del namespace in cui l'oggetto risiede (package in Java).
- name: stessa cosa come per type, può apparire fuori dai vari campi o dentro. Quando è fuori, identifica il nome dello schema. Il name e il namespace identificano univocamente lo schema. Quando si trova dentro un campo, descrive il nome del campo stesso.

I dati primitivi esistenti in Avro sono: null, int (32 bit), long(64 bit), float(single precision, 32 bit), double(double precision, 64 bit), bytes(sequenze di 8 bit), string(sequenza di caratteri unicode). I dati complessi sono 6:

- Record: una collezione di attributi. È il default e supporta tutti gli attributi sopra elencati.
- Enum: lista di elementi in una collezione. Supporta gli attributi name, namespace e symbol, che identifica i simboli del'enum come un array di nomi.
- Array: definisce un campo array con un solo attributo "items", che specifica il tipo di elemento contenuto nell'array.

- Map: identifica una mappa di elementi chiave/valore. La chiave è sempre una stringa, mentre il tipo del valore è specificato nell'attributo "values".
- Union: è usato quando un campo ha più tipi e sono rappresentati come JSON array.
- Fixed: usato per dichiarare un campo con dimensione fissa per memorizzare dati binari.

### 2.2.4.2 Serializzazione di dati

Per poter serializzare dei dati, si inizia definendo uno schema. Per semplicità, riutilizzo come esempio quello riportato nella sezione precedente. È poi necessario compilare lo schema per autogenerare la classe che ne incapsulerà le informazioni. Per farlo possiamo utilizzare sia delle istruzioni a linea di comando, "java -jar avro.jar compile schema schema.avsc destinationFolder", oppure lanciarlo tramite Ide, ad esempio Eclipse. Basta impostare le dipendenze di Avro e il plugin per la compilazione, per esempio su Maven, e lanciare "maven install", specificando l'output in cui vogliamo generare la classe. Una volta compilato, ci troviamo davanti alla classe generata da Avro. La cosa che possiamo fare è istanziarla, impostare dei valori per i campi specificati nello schema e poi iniziare il procedimento per serializzare. Creiamo un oggetto di interfaccia DatumWriter, usando SpecificDatumWriter, che converte l'oggetto Java in un formato serializzato in-memory. Successivamente, si istanzia una classe DataFileWriter per la classe Persona (quella generata da Avro nel nostro esempio), per costruire una sequenza serializzata di record di dati conformi allo schema. Si usa il metodo create() per creare un file in cui saranno memorizzati i dati che fanno match con lo schema. Si aggiungono i record creati tramite append().

```
Persona p1 = new Persona();
p1.setNome("Paolo");
p1.setEtà(23);

Persona p2 = new Persona();
p2.setNome("Mario");
p2.setEtà(30);

Persona p3 = new Persona();
p3.setNome("Giacomo");
p3.setEtà(40);

DatumWriter<Persona> perDW = new SpecificDatumWriter<Persona>(Persona.class);
DataFileWriter<Persona> perFW = new DataFileWriter<Persona>(perDW);
perFW.create(p1.getSchema(), new File("output.txt"));

perFW.append(p1);
perFW.append(p2);
perFW.append(p3);

perFW.close();

System.out.println("Serializzazione finita");
```

Figura 17. Esempio serializzazione senza parser

Per effettuare la deserializzazione, è necessario effettuare le operazioni inverse. Dato un file in cui sono serializzati i dati che vogliamo leggere, istanziamo un DatumReader parametrizzato con l'oggetto generato da Avro (Persona nel nostro esempio) e un DataFileReader con il file suddetto. Successivamente, iteriamo tramite hasNext e next sui record serializzati nel file.

```
Persona pers = null;

DatumReader<Persona> perDR = new SpecificDatumReader<Persona>(Persona.class);
DataFileReader<Persona> perFR = new DataFileReader<Persona>(new File("output.txt"), perDR);

while(perFR.hasNext()) {
    pers = perFR.next();
    System.out.println(pers.toString());
}
```

Output:

```
{"Nome": "Paolo", "Età": 23}
{"Nome": "Mario", "Età": 30}
{"Nome": "Giacomo", "Età": 40}
```

Figure 18 e 19. Deserializzazione e output

### 2.2.4.3 Serializzazione con parser

Un altro metodo per serializzare e deserializzare i record è quello di utilizzare dei parser appositi. Presentiamo subito lo stesso esempio appena visto, ma rieseguito con l'uso di parser.

```
Schema schema = new Schema.Parser().parse(new File("per.avsc"));

GenericRecord per1 = new GenericData.Record(schema);

per1.put("Nome", "Paolo");
per1.put("Età", 23);

GenericRecord per2 = new GenericData.Record(schema);

per2.put("Nome", "Mario");
per2.put("Età", 30);

GenericRecord per3 = new GenericData.Record(schema);

per3.put("Nome", "Giacomo");
per3.put("Età", 40);

DatumWriter<GenericRecord> datumWriter = new GenericDatumWriter<GenericRecord>(schema);

DataFileWriter<GenericRecord> dataFileWriter = new DataFileWriter<GenericRecord>(datumWriter);
dataFileWriter.create(schema, new File("output_parser.txt"));

dataFileWriter.append(per1);
dataFileWriter.append(per2);
dataFileWriter.append(per3);
dataFileWriter.close();

System.out.println("Serializzazione finita");
```

Figura 20. Esempio serializzazione con parser

È sufficiente create un'istanza di Schema, tramite la chiamata di parsing del file avsc al Parser di Avro. Per ogni record, si crea un'istanza di GenericRecord da GenericData. Successivamente, si effettuano le stesse operazioni dell'esempio scorso, tranne per il fatto che non abbiamo più uno SpecificDatumWriter ma un GenericDatumWriter.

```
GenericRecord p = null;
DatumReader<GenericRecord> persDR = new GenericDatumReader<GenericRecord>(schema);

DataFileReader<GenericRecord> dataFileReader =
    new DataFileReader<GenericRecord>(new File("output_parser.txt"),persDR);

while(dataFileReader.hasNext()) {
    p = dataFileReader.next();
    System.out.println(p);
}
```

Figura 21. Continuo esempio serializzazione con parser

Per effettuare la deserializzazione, ci appoggiamo ad un'istanza GenericRecord. Istanziamo un DatumReader di GenericRecord, passando il nostro schema come argomento. Istanziamo un

DataFileReader sempre di GenericRecord passando il file da cui vogliamo deserializzare i dati. Successivamente, effettuiamo iterazioni sul reader tramite hasNext e next, ottenendo i dati deserializzati.

### 2.2.5 Kafka Schema Registry

Nel momento in cui le applicazioni che si sviluppano stanno producendo o consumando messaggi per/da Kafka, possono succedere due cose:

- Nuovi consumatori di un topic esistente possono essere messi in esecuzione, consumatori che possono anche essere state sviluppate da un team diverso e in contesti di utilizzo diversi. La conseguenza è che queste nuove applicazioni devono riuscire a capire il formato dei messaggi nel topic.
- Lo schema del mondo reale cambia sempre, quindi anche lo schema dei messaggi deve cambiare di conseguenza.

Per risolvere questa problematica, è necessario avere un metodo per concordare uno schema tra tutte le applicazioni esistenti che sfruttano un particolare topic, sia esistenti sia nuove, sia a seguito di cambiamenti degli oggetti di dominio. Kafka Schema Registry si offre come soluzione.

Schema Registry è un processo server standalone che esegue in una macchina esterna ai Kafka broker. Quindi, si comporta come un producer/consumer alla vista di Kafka, anche se in realtà si occupa di mantenere un database con tutti gli schema dei topic del cluster per cui è responsabile. Questi schema sono memorizzati in Kafka topic e ne viene fatto il caching nello Schema Registry. Il Registry può essere lanciato in modalità HA e fault tolerant.

Esistono delle API con cui i consumer e producer possono accedere alle informazioni dello Schema Registry e capire se lo schema è compatibile con la versione precedente o comunque quella che loro si aspettano. Quando un producer vuole emettere un messaggio, chiama una REST API all'endpoint Schema Registry e presente lo schema del nuovo messaggio. Se lo schema è lo stesso dell'ultimo messaggio, il producer ha successo; se lo schema è diverso ma è comunque compatibile con le regole definite per quel topic, il producer ha comunque successo; se, invece, lo schema è diverso e incompatibile con le regole, il producer fallisce. Questo fallimento può essere intercettato e gestito a dovere. Se un consumer legge un messaggio che ha uno schema incompatibile con quello che il consumer si aspetta, lo Schema Registry dirà al consumer di non leggere quel messaggio.

Vengono supportati tre tipologia di formati: JSON, Avro e Protobuf.

### 2.2.5.1 Architettura

Come già citato, Kafka Schema Registry esegue al di fuori dei Kafka brokers ed è un componente addizionale che può essere configurato o meno in un qualunque Kafka cluster. Può anche essere usato da molteplici cluster. Memorizza un archivio con relative versioni degli schema esistenti e concede l'evoluzione degli schema in accordo alle regole di compatibilità definite. Offre dei serializzatori che si collegano ai client Kafka che gestiscono la memorizzazione e il recupero dello schema per i messaggi che sono inviati in un formato supportato.

I producer e consumer comunicano ancora con Kafka, ma anche con Schema Registry per mandare e recuperare gli schema che descrivono la struttura dei messaggi.

Ogni schema è associato ad un topic e ha un identificatore univoco e un numero di versione. L'ID evita l'overhead di dover inserire lo schema in ogni messaggio. Quando un producer emette un evento, viene eseguita la ricerca nello Schema Registry: se lo schema è nuovo, viene registrato e gli viene assegnato un nuovo ID univoco; altrimnti, viene restituito il suo ID. In ogni caso, l'ID viene inserito insieme all'evento e inviato al consumatore, il quale, quando incontra un evento con un ID, lo utilizza per cercare il suo schema e utilizza lo schema stesso per deserializzare i dati.

Quando si modifica uno schema è necessario prevedere diverse problematicha: informare prima i consumer o i producer, come i consumer possono gestire vecchi eventi che sono ancora memorizzati in Kafka, quanto bisogna aspettare prima che i consumer siano aggiornati.

Dopo che lo schema iniziale è stato definito, è possibile che le applicazioni evolvano e i dati debbano evolvere conseguentemente, ovvero gli schema. Quando uno schema viene cambiato, non devono "rompersi" i consumer. Dalla prospettiva di Kafka, un cambiamento di uno schema avviene quando un consumer effettua una deserializzazione, quindi una lettura. Se lo schema del consumer è diverso e da quello del producer, allora avviene una modifica automatica durante la deserializzazione per rendere conforme la lettura con lo schema del consumer, se possibile.

Per poter cambiare senza "rompere" i consumer è offerto un sistema di compatibilità di tipi, definito per uno schema. Lo Schema Registry ne supporta quattro tipologie, più tre varianti:

- BACKWARD: i consumer che usano il nuovo schema possono leggere i dati prodotti dall'ultimo schema. Esempio: la rimozione di un campo è un backward compatibility, in quanto il nuovo schema può leggere ancora quello vecchio, tralasciando il campo rimosso.
- BACKWARD_TRANSITIVE: stesso di backward, ma i consumer che usano il nuovo schema possono leggere dati prodotti da un qualunque altro schema precedente.

- FORWARD: dati prodotti dal nuovo schema possono essere letti da consumer che usano l'ultimo schema. Esempio, aggiunta di un campo.
- FORWARD_TRANSITIVE: come forward, ma i dati prodotti dal nuovo schema possono essere letti da consumer che usano qualunque schema precedente.
- FULL: il nuovo schema è sia forward che backward con l'ultimo schema registrato.
- FULL_TRANSITIVE: come full, ma con ogni schema precedentemente registrato.
- NONE: non ci sono check sulla compatibilità.

Esempi di utilizzo e di evoluzione degli schema:

- Con BACKWARD o BACKWARD_TRANSITIVE: non ci sono garanzie con cui i consumer che usano un vecchio schema possano leggere i dati prodotti con il nuovo schema. Di conseguenza è necessario dare l'upgrade di tutti i consumer prima di iniziare a produrre i nuovi messaggi.
- Con FORWARD o FORWARD_TRANSITIVE: non c'è garanzia che i consumer che usano il nuovo schema possano leggere dati prodotti da schema vecchi. Pertanto è necessario fare l'upgrade dei producer a usare tutti il nuovo schema e assicurarsi che i dati già prodotti con lo schema vecchio non siano disponibili ai consumer; successivamente si effettua l'upgrade dei consumer.
- Con FULL o FULL_TRANSITIVE: ci sono garanzie che i consumer che usano schema vecchi possano leggere dati prodotti da schemi nuovi e che consumer che usano schemi nuovi possano leggere dati prodotti da schema vecchi. Pertanto è possibile fare l'upgrade di consumer e producer in maniera indipendente.
- Con NONE: necessità di forti attenzioni quando si aggiornano consumer/producer, in quanto non si ha nessun check.

Il default è BACKWARD.

Il check può fallire se:

- Il producer aggiunge una colonna obbligatoria e i consumer usano BACKWARD o FULL.
- Il producer elimina una colonna obbligatoria e i consumer usano FORWARD o FULL.

Il check ha successo se:

- Il producer aggiunge una colonna obbligatoria e i consumer usano FORWARD.
- Il producer aggiunge un campo opzionale e i consumer usano BACKWARD.
- Il producer elimina un campo opzionale e i consumer usano FORWARD o FULL.

Per accedere alle informazioni e gestire gli schema dello Schema Registry è possibile utilizzare le Kafka REST API, che consentono di effettuare le seguenti operazioni:

- Memorizzare gli schema per chiavi e valori dei record.
- Lista degli schema per soggetto
- Lista di tutte le versioni degli schema di un soggetto
- Ottenere uno schema dalla versione
- Ottenere uno schema dall'ID
- Ottenere l'ultima versione di uno schema
- Effettuare un check di compatibilità
- Impostare il livello di compatibilità globale

### 2.2.6 Kafka Streams

Kafka Streams è una libreria nata per fare il building di applicazioni e microservizi, dopo gli input e gli output confluiscono in un Kafka Cluster. Combina la semplicità di scrivere e fare il deploy di standard Java/Scala con i benefici della tecnologia di Apache Kafka. I principali benefici di Kafka Streams sono:

- Semplice e leggera libreria client: può essere inserita in un'applicazione Java, per poter essere integrata come un qualunque altro package esistente.
- Dipendenze: non ha dipendenze esterne di sistemi, se non con Apache Kafka stesso come messaging layer. Usa il modello di partizionamento di Kafka per scalare orizzontalmente e mantenendo forti garanzie di ordinamento.
- Fault tolerant: supporta tolleranza di errori per lo stato locale, cosa che consente operazioni con stato molto veloci ed efficienti, come join e aggregazioni con finestre (windowing).
- Supporta la semantica exactly-once, per garantire che ogni record sia processato solo una volta, anche in presenza di errori.
- Sfrutta elaborazione di un record alla volta, ottenendo una latenza dell'ordine di millisecondi. Supporta operazioni di windowing basate sul tempo degli eventi con arrivo di record fuori ordine.
- Offre primitive necessarie per lo stream processing, sia con high-level API (Kafka Streams DSL) che con low-level Processor API.

## 2.2.6.1 Topologia dello Stream Processing

Lo Stream è l'astrazione più importante offerta da Kafka Streams: rappresenta un illlimitato e continuamente aggiornato insieme di dati. Uno stream è un'ordinata, riproducibile e fault-tolerant sequenza di record immutabili, dove un record è definito come una coppia chiave-valore. Uno Stream processing application è un qualunque programma che usa le librerie di Kafka Streams. Definisce la propria logica computazionale attraverso una o più topologie di processori, le quali sono un grafo di stream processor (nodi) che sono connessi dagli stream (archi). Uno Stream Processor è un nodo nella processor topology: rappresenta uno step di processamento in cui si trasformano i dati in stream, ricevendo un record in input alla volta dai suoi processori a monte nella topologia, applicando eventuali operazioni e, eventualmente, producendo uno o più record in output sui processori a valle nella topologia.

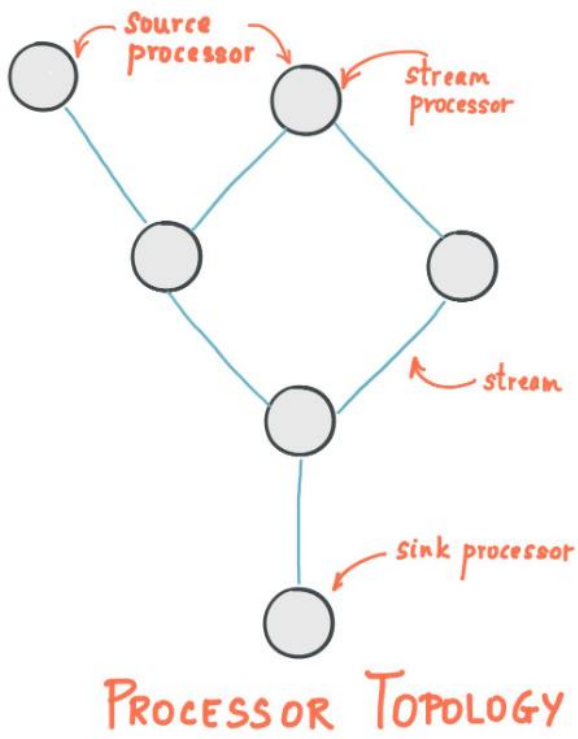


Figura 21. Topologia di Kafka Streams

Esistono due tipologie di processori:

- Source Processor: non ha nessun processore a monte (prima di lui) nella topologia. Produce un input stream nella sua tipologia da uno o più topic consumando i record da questi topic e inoltrandoli ai suoi processori successivi (a valle).
- Sink Processor: non ha nessun processore a valle (dopo di lui) nella topologia. Manda ogni record che riceve dai suoi processori a monte verso uno specifico topic.

Kafka Streams espone due tipologie di implementazione della topologia di processamento:

- Kafka Streams DSL: offre le più comuni operazioni di trasformazione, come map, filter, join e aggregations.
- Processor API: concede agli sviluppatori di definire e connettere processori custom in modo da interagire con "state stores".

### 2.2.6.2 Tempo

Un aspetto critico in uno stream processing è la nozione di tempo e come è modellato e integrato. Per esempio, alcune operazioni come il windowing sono definite tramite vincoli di tempo. I più comuni concetti di tempo negli stream sono:

- Event time: il punto temporale in cui un evento o un record avviene/si crea dalla sorgente. Per esempio, se l'evento è il cambiamento della posizione GPS di una macchina, l'event time sarà l'istante di tempo in cui il sensore GPS ha catturato il cambiamento di posizione.
- Processing time: punto temporale in cui un evento o un record inizia ad essere processato dall'applicazione di stream processing, per esempio quando un record viene consumato. Il processing time sarà millisecondi, minuti, ore, giorni ecc dopo l'event time.
- Ingestion time: Istante di tempo in cui un evento o record viene memorizzato in una partizione di un topic dal Kafka Broker. La differenza con l'event time è che nell'ingestion time, il timestamp è applicato quando il record viene inserito nel topic target dal broker, non quando l'evento è creato dalla sorgente.

La scelta tra event time e ingestion time è effettuata dalle configurazioni di Kafka, non Kafka Streams. Da Kafka 0.10.x in poi, i timestamp sono inglobati direttamente nei messaggi Kafka. A seconda della configurazione di Kafka, tale timestamp identifica l'event time o l'ingestion time. La configurazione può essere fatta o per ogni topic o per ogni broker. I timestamp vengono assegnati da Kafka tramite l'interfaccia TimestampExtractor.

### 2.2.6.3 Dualità tra Stream e Table

Quando implementiamo stream processing nella pratica, abbiamo tipicamente bisogno sia di stream che di database. Immaginiamo un caso comune di un sito di e-commerce: si ha uno stream di transazioni dei clienti che vengono arricchiti con informazioni sui clienti stessi, residenti in un database. Kafka Streams garantisce quindi delle API per gestire Stream e Table. Un'importante considerazione che possiamo fare è che esiste una forte correlazione tra stream e table, chiamata dualità tra stream e table. Kafka offre questa dualità in diversi modi: per esempio, per avere un'applicazione elastica, per supportare fault-tolerant stateful processing o per lanciare query interattive verso gli ultimi risultati dello stream processing. Oltre all'utilizzo interno che ne fa Kafka, questa dualità viene esposta anche all'esterno, agli sviluppatori.

| key1 | value1 |
|---|---|
| key2 | value2 |
| key3 | value3 |

Figura 22. Record di esempio

La dualità dice essenzialmente che una table può essere vista come uno stream e viceversa. Una semplice rappresentazione di una table può essere un'insieme di coppie chiave-valore, chiamata anche mappa (map).

La dualità tra stream e table descrive la forte relazione tra stream e table:

- Stream as Table: uno stream può essere considerato un changelog di una table, dove ogni record nello stream cattura uno stato di cambiamento della table. Uno stream è una table sotto mentite spoglie e può facilmente essere trasformata in una table reale, riproducendo il changelog dall'inizio alla fine per ricostruire la table. Quindi, aggregando record in uno stream daremo vita a una table.

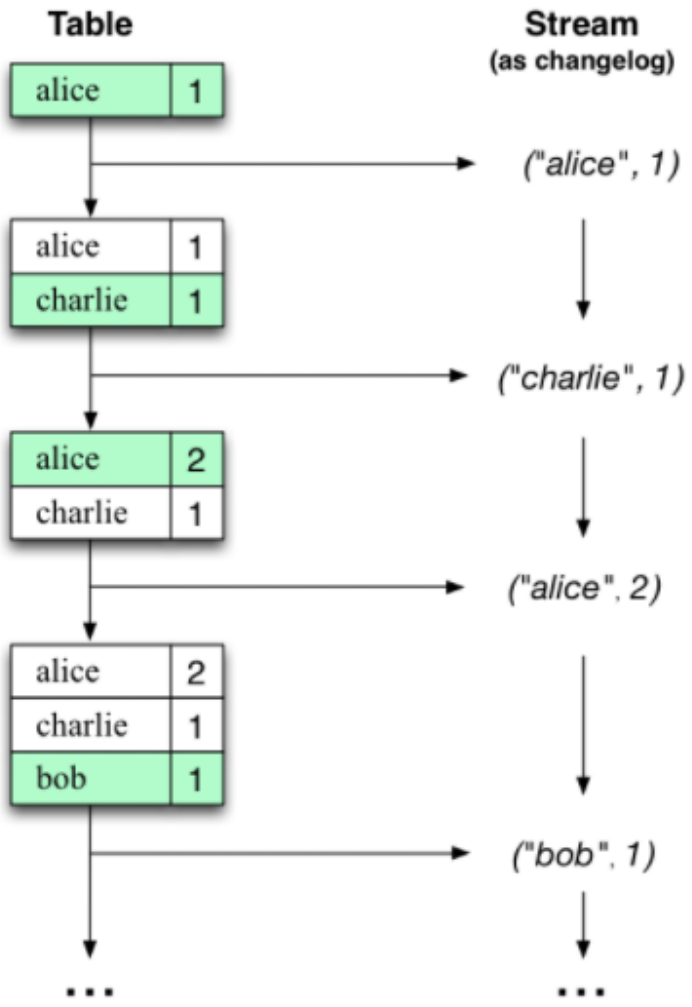


Figura 23. Da Stream a Table

- Table as Stream: Una table può essere considerata uno snapshot, in un certo istante di tempo, dell'ultimo valore di ogni chiave di uno stream. La table è, a sua volta, uno stream sotto mentite spoglie e può facilmente essere trasformato in uno stream reale iterando in ogni coppie chiave-valore che fa parte della table.

Immaginiamo di avere una table che tiene traccia del numero totale di visitatori di una pagina web da degli utenti. In generale, ogni volta che un evento di visualizzazione pagina è processato, lo stato della table viene aggiornato correttamente. Lo stato cambiato tra diversi istanti di tempo può essere rappresentato come un changelog.

Allo stesso tempo, grazie alla dualità tra stream e table, lo stesso stream può essere usato per ricostruire la tabella originale.

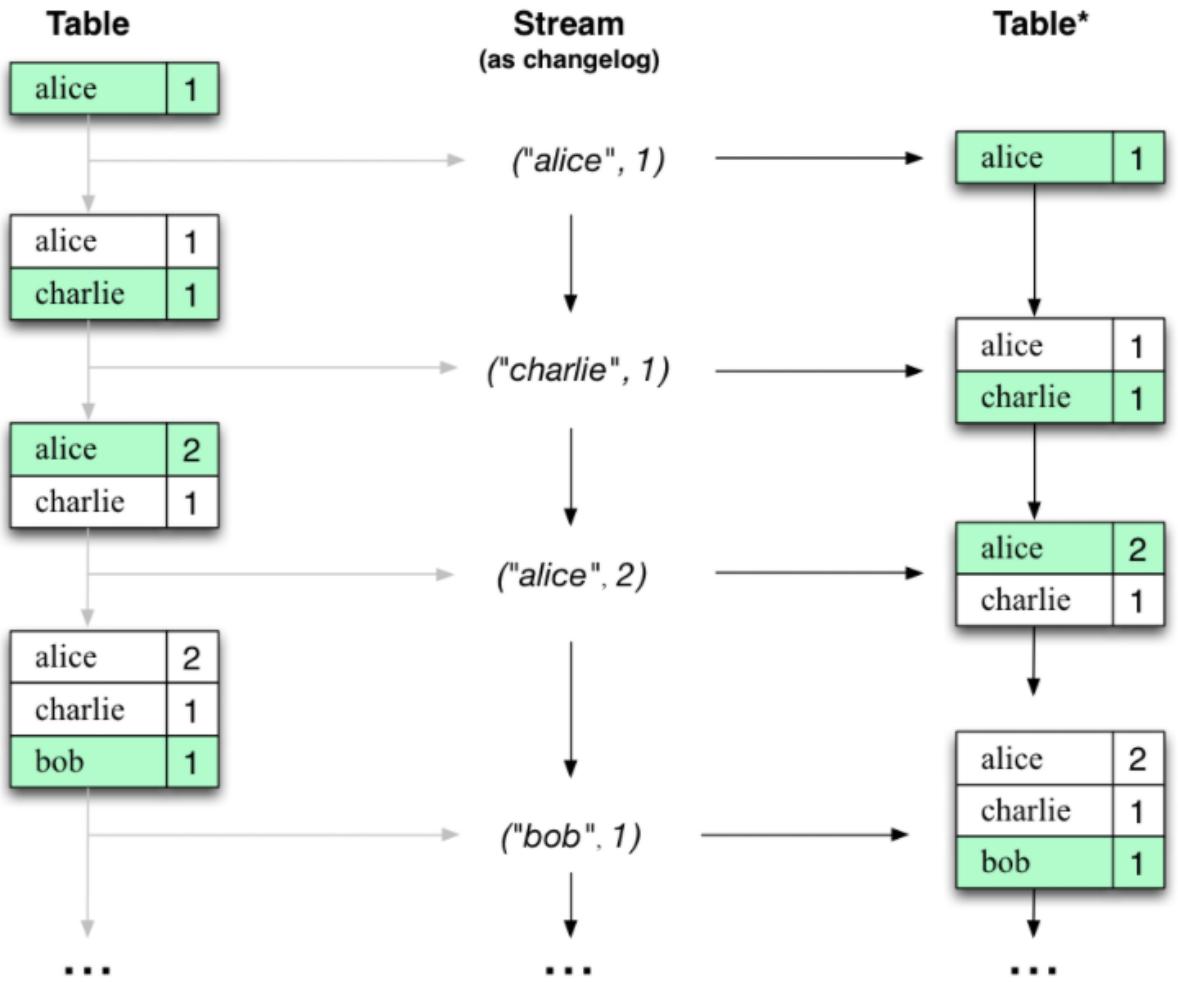


Figura 24. Dualità tra Stream e Table

## 2.2.6.4 Aggregazione

L'aggregazione è un'operazione che prende in input uno stream o una table e produce in output una nuova table, combinando molteplici record di input in un singolo record di output. Esempi di aggregazione sono calcolo di conteggi o di somme. Nel Kafka Streams DSL un input stream per un'aggregazione può essere una KStream o una KTable, mentre l'output stream sarà sempre un KTable. Creare una table come output fa sì che, anche se arrivano record dopo la produzione del valore, è possibile aggiornare a posteriori l'aggregato. Se accade che arriva un record dopo la produzione, l'aggregante emetterà un nuovo valore aggregato, che sostituirà il vecchio valore (visto che come output si ha una KTable).

### 2.2.6.5 Windowing

L'operazione di windowing ci fa controllare come raggruppare record che hanno la stessa chiave per operazioni stateful, come aggregazione o join, in cosiddette finestre (windows). Le finestre sono tracciate tramite la chiave del record. In Kafka Streams DSL sono presenti le operazioni di windowing. Quando lavoriamo con finestre, possiamo specificare un grace period per la finestra. Questo grace period controlla quanto a lungo dovrà attendere Kafka Streams per record che arrivano fuori dalla finestra data. Se un record arriva dopo che il grace period di una finestra è passato, sarà scartata e non processato in tale finestra. Nello specifico, un record viene scartato se il relativo timestamp indica che fa parte di una certa finestra, ma lo stream time è maggiore della fine della finestra più il grace period.

### 2.2.6.6 Stati

Alcune applicazioni di stream processing non richiedono lo stato, che significa che il processamento di un messaggio è indipendente dal processamento di tutti gli altri messaggi. Potrebbe essere utile in alcuni contesti riuscire a mantenere stato, in quanto apre diverse possibilità interessanti: potremmo fare join di input stream o raggruppare o aggregare record. Diverse operazioni con stato sono offerte da Kafka Streams DSL. Viene data la possibilità di gestire le cosiddette state stores, che possono essere usate dalle applicazioni di stream processing per immagazzinare e interrogare i dati. Ogni task in Kafka Streams ingloba uno o più state stores che possono essere accedute tramite API. Questi state store possono essere sia dei persistenti insiemi di coppie chiave-valore, sia un hashmap in memoria o qualunque altra struttura dati che conviene in un determinato contesto. Kafka concede la diretta esecuzione di query read-only sugli state store. Questo fa in modo di avere l'opportunità di effettuare le cosiddette query interattive. Tutti gli store sono denominati e le query interattive espongono solo le operazioni di lettura dell'implementazione sottostante.

### 2.2.6.7 Garanzie di processamento

Ci troviamo spesso di fronte ad avere bisogno di una semantica "exactly-once" in applicazioni di stream processing. Prima di kafka 0.11.0.0, era prevista solo la semantica at-least-once, senza

possibilità di garantire una “exactly-once” end-to-end. Ma dalla versione 0.11.0.0, è stato aggiunto il supporto che garantisce ai producer di mandare messaggi a differenti partizioni di un topic in un modo transazionale e idempotente, oltre ad aggiungere la semantica “exactly-once”. Dalla versione 2.6.0, tale semantica è stata utilizzata in termini di utilizzo di risorse e connettività di rete.

### 2.2.6.8 Gestione dei record in arrivo fuori ordine

Una problematica che potrebbe dover gestire un’applicazione di stream processing è quella di gestione di record in arrivo fuori ordine, che può impattare la logica di business. In Kafka Streams possono esserci due potenziali cause per questo, in rispetto ai timestamp:

- In una partizione di un topic: un timestamp di un record può non essere monotono crescente rispetto agli offset. Poiché Kafka prova a processare record in una partizione di un topic seguendo l’ordine dell’offset, può succedere che un record con timestamp più grande (ma con offset più piccolo) sia processato prima di un record con timestamp più piccolo (ma con offset più grande) nella stessa partizione di un topic.
- In uno stream task: se il task sta processando molteplici partizioni in un topic e un utente configura l’applicazione in modo da non aspettare che tutte le partizioni contengano dei dati bufferizzati e prende dalla partizione con il minor timestamp per processare il prossimo record, allora quando altri record sono fetchati da altre partizioni di un topic, i loro timestamp potrebbero essere più piccoli di quelli già processati precedentemente.

Per le operazioni stateless, i record fuori ordine non impattano con la logica di processing, in quanto solo un record è processato in ogni istante di tempo, senza guardare alla storia dei processamenti passati. Per le operazioni stateful, potrebbe invece essere un problema. Per quanto riguarda i join, gli utenti devono essere consapevoli che alcuni dei record fuori ordine non possono essere gestiti aumentando la latenza e il costo negli stream:

- Per Stream-Stream join, tutti i tipi (inner, outer, left) possono gestire correttamente i record fuori ordine, ma lo stream di output potrebbe contenere left record-null non necessari per left join e left record-null o null-rightrecord per outer join.
- Per Stream-Table join, i record fuori ordine non sono gestiti e possono addirittura provocare risultati non predicibili.
- Per Table-Table join i dati fuori ordine non sono gestiti. Comunque, il risultato sarà un changelog, con proprietà di “eventually consistent”.

## 2.2.6.9 Kafka Streams API Stateless

Cerchiamo di capire come può essere sviluppata un'applicazione con Kafka Streams, utilizzando Kafka Streams DSL. I pattern di utilizzo sono solitamente molto simili, ma iniziamo da un approccio il più semplice e intuitivo possibile.

- Si definiscono gli oggetti di configurazione.
- Si creano le istanze di Serdes, sia predefinite che custom. Serdes sono le entità che permettono la serializzazione e deserializzazione degli oggetti che dobbiamo gestore con Kafka Streams.
- Si costruisce la processor topology
- Si crea e si fa partire il KStream.

Il primo step per creare un'applicazione Kafka Streams è definire un node sorgente (source node), il quale è responsabile di consumare i record da un topic e trasmetterli all'applicazione.

```
KStream<String, String> myStream = builder.stream("input-topic"),
    Consumed.with(stringSerde, stringSerde));
```

In questo esempio, abbiamo creato un'istanza che consuma messaggi scritti nel topic "input-topic". Specifichiamo anche la tipologia di oggetti che sono gestiti da questo topic, in modo tale da informare Kafka Streams su come fare la serializzazione e deserializzazione dei dati, tramite Consumed.

```
KStream<String, String> intermediateStream = myStream.mapValues(String::toLowerCase);
```

Qua abbiamo aggiunto un ulteriore processor, ovvero un nodo figlio del processor precedente (che faceva da sorgente). Chiamanto KStream.mapValues, stiamo creando un nuovo nodo i cui input sono i risultati dell'esecuzione della chiamata mapValues. Non si dovrebbe modificare il valore originale nel ValueMapper fornito a mapValues. Lo stream intermediateStream riceve una copia trasformata del valore iniziale della chiamata a myStream.mapValues, nel caso specifico il testo portato a lowercase. Un'importante osservazione al parametro della chiamata mapValues: è un'istanza dell'interfaccia ValueMapper<V,V1>, che definisce solo il metodo "apply". Questo fa sì che si possano utilizzare lambda expressions come parametro, rendendo molto agevole l'utilizzo di questa API. In questo caso, avremmo infatti potuto scrivere "str -> str.toLowerCase()", ma va bene anche il riferimento al metodo.

```
intermediateStream.to("output-topic",Produced.with(stringSerde, stringSerde);
```

Il metodo KStream.to crea un sink processor, cioè un nodo che scrive i dati indietro a Kafka, ultimo nodo della topologia. In questo caso, li scrive nel topic “output-topic”.

Kafka Streams è altamente configurabile: infatti, è possibile configurare molte proprietà, al fine di perfezionare il tutto a seconda dei requisiti di contesto applicativo. Due proprietà che devono essere sempre configurate sono: APPLICATION_ID_CONFIG e BOOTSTRAP_SERVERS_CONFIG. La prima identifica in maniera univoca l’applicazione Kafka Streams all’interno di un cluster.Viene anche utilizzata come prefisso per il client ID e per il group ID, se non specificato altrimenti. La seconda può rappresentare un singolo hostname:port oppure una molteplicità di hostname:port separati da virgola. Il valore dà modo di avere un’idea di dove siano i Kafka cluster e come comunicarci.

```
StreamsConfig.APPLICATION_ID_CONFIG
```

Un altro aspetto è quello di Serde. Come accennato precedentemente, i Serde sono istanze necessarie per la serializzazione e deserializzazione dei dati, utilizzati da Kafka Stream sia in modalità di default, con le implementazioni del tipo String, Long, Integer, Double e Byte Array, sia in modalità custom, in cui lo sviluppatore può creare le propriè funzionalità per ogni tipologia di dato di cui ha bisogno.

```
Serde<String> stringSerde = Serdes.String();
```

Per creare un Serde custom, è possibile utilizzare le interfacce Serializer<T> e Deserializer<T> fornite da Kafka Streams. Immaginiamo di voler creare un Serde per oggetti JSON, è necessario dire a Kafka come convertire un oggetto prima in formato JSON e poi in un byte array, quanto tale dato viene mandato a un topic (i dati ai topic sono mandati sotto forma di byte). Al tempo stesso, prelevando i dati da un topic, è necessario dire a Kafka come deserializzare i byte in arrivo in oggetti JSON, andando a ritroso. L’idea è di implementare, per la serializzazione, l’interfaccia Serializer<T> e andare a fare override del metodo serialize, costruendo un serializer ad hoc.

```
@Override
public byte[] serialize(String topic, T data){
    return gson.toJson(data).getBytes(Charset.forName("UTF-8"));
}
```

Stessa cosa per la deserializzazione, in cui però facciamo override del metodo deserialize.

Una delle funzionalità base che Kafka Streams mette a disposizione è la possibilità di effettuare dei filtraggi nei dati in input, utilizzando filter. Immaginiamo di avere uno stream di stringhe e di voler filtrare solo le stringhe che abbiano lunghezza in caratteri maggiore di 5.

```
KStream<String, String> stream = builder.stream(INPUT_TOPIC);
stream.filter((k, v) -> v.length() > 5).to(OUTPUT_TOPIC);
```

Tramite filter, possiamo specificare un parametro, che sarà un Predicate, espresso in questo caso da una lambda expression, che filtra i valori della stringa con lughezza maggiore di 5.

Immaginiamo di dover effettuare uno split di uno stream in due stream che possono scrivere in topic differenti.Questo può essere reso possibile dal metodo KStream.branch: prende un numero arbitrario di Predicate e restituisce un array di KStream, la cui dimensione fa match con il numero di predicati passati come parametro.

```
Predicate<String, Item> isCPU = (k,v) -> v.getType().equals("CPU");
Predicate<String, Item> isGPU = (k,v) -> v.getType().equals("GPU");

int cpu = 0;
int gpu = 1;

KStream<String, Item>[] branch = myStream.branch(isCPU, isGPU);

branch[cpu].to("cpus",Produced.with(stringSerde,itemSerde);
branch[gpu].to("gpus",Produced.with(stringSerde,itemSerde);
```

Nell'esempio mostrato, si suppone di avere uno stream di item di un magazzino. Si vuole separare in dipartimenti diversi gli stream relativi a CPU e GPU. Per farlo, è possibile creare dei predicati che filtrino il tipo di item e passarli come parametro al metodo branch, applicato allo stream originale. Successivamente, è possibile accedere all'array di stream creato e mandare ciascuno in topic diversi.

Kafka utilizza come formato di messaggi delle coppie chiave/valore, ma non ci sono vincoli sul fatto che la chiave debba essere non nulla. Nella pratica, se non c'è bisogno di una chiave, allora renderla nulla e quindi non averla riduce l'overhead e lo spreco di risorse. Supponiamo di avere, quindi, uno stream senza chiavi. Se ci troviamo di fronte alla necessità di mandare i dati di questo stream verso un NoSQL database che sfrutta come memorizzazione il formato chiave/valore, possono nascere delle problematiche. Il metodo KStream.selectKey ci aiuta proprio in questo: restituisce una nuova istanza di KStream che produce dei record con una nuova chiave e stessi valori dello stream in ingresso. Per la creazione della nuova chiave, è possibile passare una lambda expression come parametro.

Un'ultima considerazione sulle API stateless può essere fatta sul metodo foreach. È un nodo processor che semplicemente offre la possibilità di usare un ForeachAction per effettuare l'azione definita in tale oggetto so ogni record che riceve. È quindi necessario definire un ForeachAction con le azioni da fare, per poi passarlo come parametro a KStream.foreach di uno stream.

## 2.2.6.10 KStream Stateful API

Fino ad ora, abbiamo introdotto le API per una gestione stateless delle operazioni. Infatti, consideravamo ogni transazione come a sé stante, isolata dalle altre. Non abbiamo considerato eventuali eventi che potevano verificarsi nello stesso tempo di un'operazioni, oppure in un intervallo di tempo prima o dopo l'operazione stessa. Inoltre, non abbiamo neanche accennato al fatto che più stream possono essere uniti (join). La prima cosa che è necessario introdurre è il concetto di stato. Lo stato non è altro che l'abilità, da parte di un'applicazione, di ricordarsi cosa è successo in passato e farne match con i dati attuali.

Parlare di stato in un'applicazione di stream processing può sembrare strano a primo impatto: infatti, lo stream processing si basa sul continuo flusso di dati o eventi che non hanno quasi niente a che fare l'un con l'altro. Possiamo vedere inizialmente lo stato un po' come una tabella di un database, ma in uno stream processing cambia molto più velocemente. Inoltre, non deve necessariamente avere stato per lavorare con stream processing: spesso l'evento stesso o i dati che arrivano sono autoesplicativi, senza necessità di memorizzare nient'altro. Ciononostante, talvolta può fare comodo sfruttare questo concetto anche nello stream processing, sia per eventi verificati in precedenza sia per effettuare join di stream.

La base delle funzioni stateful in Kafka Streams è il metodo KStream.transformValues. Questo metodo fa semanticamente la stessa cosa di mapValues, ma con un po' di differenze. TransformValues ha accesso a uno StateStore per portare a termine il proprio compito. Inoltre, ha l'abilità di fare lo scheduling di operazioni che possono avvenire a intervalli regolari tramite il metodo punctuate. Il metodo transformValues prende come parametro un ValueTransformerSupplier<V,R>, che fa da supplier di un'istanza di ValueTransformer<V,R>. La cosa da fare, quando è necessario utilizzare tale metodo, è predisporre una propria implementazione di ValueTransformer. Per prima cosa, bisogna definire il metodo init(), in cui si va a prendere lo StateStore creato al momento del build della topologia.

```
private KeyValueStore<String, String> stateStore;
private String storeName;
private ProcessorContext context;

public void init(ProcessorContext context){
    this.context=context;
    stateStore=(KeyValueStore) this.context.getStateStore(storeName);
}
```

Immaginiamo di dover tenere traccia del totale di acquisti di un cliente in un certo negozio. Sfruttiamo un approccio stateful al fine di memorizzare in uno StateStore le informazioni che vogliamo tenere in considerazione. Ciò che facciamo è fare una lookup nello stato, al momento del bisogno in un punto della topologia, con l'ID del cliente. Successivamente dovremmo aggiornare lo stato del dato attuale e portarlo avanti nella topologia. Ma è necessario capire una cosa: poiché le transazioni vanno dentro l'applicazione senza una chiave, ma noi avremmo bisogno di garantire che per ogni cliente le transazioni siano sulla stessa partizione, in realtà se procediamo senza fare nient'altro, ciò che accadrebbe è che verrebbe applicata la politica round robin per l'associazione delle partizioni, cosa che noi non vogliamo. Infatti, avere dati su più partizioni significherebbe anche accedere a molteplici StateStore per ogni transazione (perché per ogni partizione è associato uno StreamTask, il quale ha un univoco StateStore a sé stante). Il modo per risolvere questa problematica è utilizzare la ripartizione dei dati per ID del cliente.

La ripartizione dei dati è molto intuitiva a livello astratto: per ripartizionare, è necessario prima di tutti associare/modificare la chiave del record originale, poi scriverlo su un nuovo topic. Alla fine, rileggiamo da quel topic i record, che originariamente potevano venire anche da partizioni diverse, ma adesso, se utilizziamo una chiave, vengono raggruppati secondo l'hash della chiave modulo numero delle partizioni, ovvero ogni record con la stessa chiave andrà nella stessa partizione.

In Kafka Streams è possibile sfruttare il metodo KStream.through(), il quale crea un topic intermedio e il KStream corrente inizierà a scrivere su di esso. Successivamente, è restituita una nuova istanza di KStream, dopo l'esecuzione del metodo, usando il topic intermedio come sorgente.

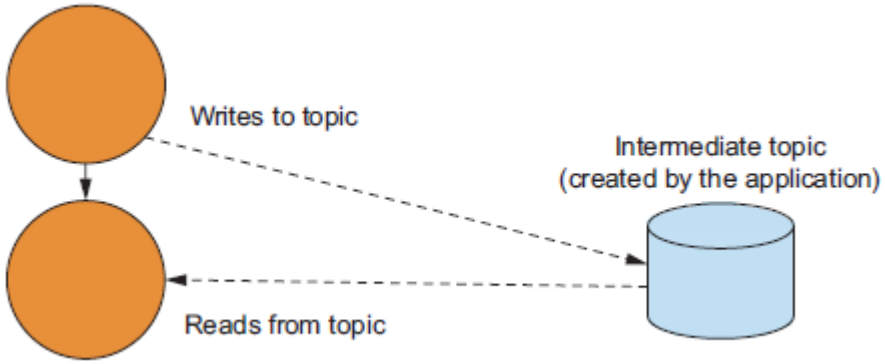


Figura 25. Kafka Streams Through

Se le chiavi sono state modificate, non c'è bisogno di una strategia custom di partitioning, basta utilizzare il DefaultPartitioner per gestire il ripartizionamento. Altrimenti, è possibile implementare un proprio partitioner, tramite StreamPartitioner, per esempio basandosi sull'ID dei clienti. Per farlo, basta implementare StreamPartitioner e definire il metodo partition, restituendo la partizione voluta

a seconda della strategia più consona al contesto in cui ci troviamo.

```
KStream<String, Purchase> transformByIdStream =
    myStream.through("customers",Produced.with(stringSerde, purchaseSerde, streamPartitioner));
```

Esistono due principi fondamentali che possono impattare sulla performance e affidabilità dell'applicazione Kafka Streams:

- Località dei dati: la località è cruciale per la performance. Se dovessimo accedere a database remoti per accedere alle informazioni memorizzate, avremo una latenza dovuta alla rete non trascurabile, soprattutto se stiamo scalando su un enorme numero di nodi. Sarebbe opportuno avere uno StateStore locale per ogni nodo, così se un processo fallisce, non impatta su altri processi di stream processing correlati.
- Fault Tolerant: sono utilizzate dei topic di changelog per garantire un fault tolerant efficace. Nel caso in cui un nodo fallisse, poiché i dati sono stati memorizzati nel topic di changelog, quando il guardo viene riparato, i dati vengono ripristinati correttamente nello StateStore (dati recuperati fino all'ultimo commit degli offset nel changelog). Notare che questa procedura non produce troppo overhead: infatti, i record sono sempre messi in cache, bufferizzati, e inviati al topic solo dopo un flush. Questo fa sì che solo l'ultimo valore di un record per una data chiave viene scritto, non tutte le modifiche intermedie.

Cerchiamo di capire adesso come si può utilizzare uno StateStore in KafkaStreams. La prima cosa da fare è creare uno StoreSupplier tramite uno dei metodi factory della classe Stores. Ci sono due classi addizionali per configurare lo StateStore: Materialized (usato nelle high-level API di Kafka Streams DSL) e StoreBuilder (usato principalmente nelle Processor API).

```
String myStateStoreName = "myStateStore";
KeyValueBytesStoreSupplier storeSupplier =
    Stores.inMemoryKeyValueStore(myStateStoreName);
StoreBuilder<KeyValueStore<String, Integer>> storeBuilder =
    Stores.keyValueStoreBuilder(storeSupplier, Serdes.String(), serdes.Integer());

builder.addStateStore(storeBuilder);
```

Al posto di Stores.inMemoryKayValueStore, è possibile utilizzare altri metodi factory esistenti: persistentKeyValueStore, lruMap, persistentWIndowStore, persistentSessionStore.

Ogni StateStoreSupplier ha il logging abilitato a default. In questo caso, per logging si intende la creazione da parte di Kafka di un topic di changelog, per garantire fault tolerance. Il logging può essere disabilitato usando la factory Stores.disableLogging(). Questi changelog sono configurabili attraverso il metodo withLoggingEnabled(Map<String,String> config). La configurazione è

importante, è possibile gestire, per esempio, una dimensione di retention e un tempo di retention. Per far questo, basta creare una Map in cui inseriamo come chiave il nome della proprietà da configurare, per esempio "retention.ms", con il valore che vogliamo impostare, per esempio "10000". A questo punto, una volta creata la mappa, la passiamo come parametro al metodo withLoggingEnabled.

```
Map<String, String> config = new HashMap<>();
config.put("retention.ms","10000");
config.put("retention.bytes","100000");
storeBuilder.withLoggingEnabled(config);
```

Abbiamo discusso sulle due politiche di pulizia dei topic/log: compattazione ed eliminazione. A default, Kafka utilizza la compattazione. Ma se avessimo un changelog con tante chiavi uniche, la compattazione non riduce notevolmente la dimensione del topic. È quindi possibile configurare una politica di cleanup, tramite "cleanup.policy", impostandola su "compact,delete". Così, il changelog sarà mantenuto di una dimensione ragionevole, anche con tante chiavi uniche.

## 2.2.6.11 Joining

Talvolta potrebbe essere necessario avere come stato un altro stream: è possibile combinare differenti eventi da due stream con la stessa chiave per formarne uno nuovo. Immaginiamo di avere due stream senza chiave e di cui vogliamo fare il join. Come appena detto, è necessario che i due stream abbiamo la chiave e sia anche dello stesso tipo. In questo caso, dobbiamo quindi generare una chiave affinchè possiamo effettuare correttamente un join. Consideriamo come esempio i due stream che abbiamo ottenuto nell'esempio del branch: siamo partiti da uno stream myStream e abbiamo effettuato il branch su isCPU e isGPU. Adesso, dobbiamo prima generare una chiave per poter effettuare un eventuale join successivamente.

```
KStream<String, Item>[] branch = myStream
    .selectKey((k,v) -> v.getBrand()).branch(isCPU,isGPU);
```

Nel momento in cui invochiamo un metodo che genera una nuova chiave (che può essere map, selectKey o transform), viene impostato un flag interno a "true", il quale indice che il KStream appena generato richiede una ripartizione. Quando questo flag è "true", se effettuiamo un join, una reduce o un'aggregazione, il ripartizionamento è gestito automaticamente da Kafka. In questo esempio, invochiamo selectKey su myStream e successivamente effettuiamo un branch: i due stream di output hanno entrambi il flag a "true", che indicano che necessitano un ripartizionamento.

Ora che abbiamo i due stream pronti, con lo stesso tipo di chiave, possiamo capire come effettuare un join. Innanzitutto, servirà un ValueJoiner<V1, V2, R>, entità che effettua le condizioni del join: prende in input due oggetti, non necessariamente dello stesso tipo, e restituisce un terzo oggetto, anch'esso eventualmente differente dagli altri. Il ValueJoiner sarà passato come parametro al join. Notare che sarebbe possibile, soprattutto per Joiner semplici e con poche operazioni, passare direttamente una lambda expression, in quando ValueJoiner offre solo un metodo di apply, da implementare come vogliamo. Immaginiamo di aver implementato un ItemJoiner, che implementa ValueJoiner<Item, Item, ResultItem>: è quindi possibile adesso invocare l'operazione di join.

```
ValueJoiner<Item, Item, ResultItem> itemJoiner = new ItemJoiner();
JoinWindows window = JoinWindows.of(60 * 1000 * 10);

KStream<String, ResultItem> joined = cpuStream.join(gpuStream,
    itemJoiner, window, Joined.with(stringSerde, itemSerde, itemSerde));

joined.print("joined");
```

Esistono due metodi addizionali per JoinWindows, al fine di specificare l'ordine degli eventi:

- streamA.join(streamB,…,window.after(10000)…): specifica che il timestamp del record dello streamB è al massimo 10 secondi dopo il timestamp del recor dello streamA. Il limite iniziale della finestra non cambia.
- streamA.join(streamB,…,window.before(10000)…): specifica che il timestamp del record dello streamB è al massimo 10 secondi prima del timestamp del record dello streamA. Il limite finale della finestra non cambia.

Per poter effettuare un join in Kafk Streams, è necessario che i partecipanti al join stesso siano co-partizionati, ovvero devono avere lo stesso numero di partizioni ed essere sottoposti a chiave dello stesso tipo. Come risultato da questi requisiti, effettuare un join scatena in automatico il controllo sulla ripartizione degli stream coinvolti. È necessario che lo sviluppatore controlli che le chiavi siano dello stesso tipo; inoltre, è necessario che i producer utilizzino la stessa classe di partizionamento quando scrivono nei topic.

Esistono altri due tipi di join: quello che è stato affrontato adesso si chiama inner-join, gli altri sono outer join e left outer join.

- Outer join: restituisce sempre un record, ma il record di output può non includere entrambi gli eventi specificati dal join. Se solo l'evento del primo stream è disponibile nella finestra di tempo, allora solo questo sarà incluso nell'output; se sono disponibili entrambi, faranno parte

tutti e due del'output; se è disponibile solo il secondo stream, solo questo farà parte dell'output.

- Left-outer join: simile a un outer join, ma con una differenza: se l'evento è disponibile solo nel secondo stream (il primo è nullo/vuoto), allora non ci sarà output.

## 2.2.6.12 Timestamp

Abbiamo già introdotto il concetto di timestamp e differenziato tre tipologie (event time, processing time, ingestion time), ma non si è affrontato come gestirli nella pratica. Per abilitare differenti semantiche di tipologie di timestamp, Kafka Streams offre un'interfaccia, TimestampExtractor, con quattro implementazioni già predefinite. Quasi tutte le implementazioni di TimestampExtractor lavorano con timestamp impostati dal producer o dal broker, offrendo quindi semantica event time e log-append-time (broker). ExtractRecordMetadataTimestamp è una classe astratta che offre le funzionalità chiave per estrarre i timestamp dai metadati dei record. È necessario implementare onInvalidTimestamp, per gestire eventuali timestamp non validi (per esempio minori di 0). Un'altra tipologia di estrattore di timestamp è WallclockTimestampExtractor, che in realtà non estrae timestamp, ma restituisce il tempo in millisecondi chiamando System.currentTimeMillis().

È possibile creare dei custom TimestampExtractor. Per farlo, è necessario implementare TimestampExtractor, con il metodo chiave "public long extract". Ci sono due modi per impostare l'utilizzo di un TimestampExtractor custom: il primo è quello di configurarlo a livello globale tramite configurazione iniziale, sempre con una proprietà di StreamsConfig, DEFAULT_TIMESTAMP_EXTRACTOR_CLASS. La seconda opzione è passare un'istanza dell'estrattore custom tramite Consumed. Il vantaggio è di avere un TimestampExtractor per ogni sorgente di input, con possibilità di differenziare quindi ogni contesto di record.

## 2.2.6.13 KTable API

Finora abbiamo discusso riguardo a uno stream di dati, utilizzando quindi una vista ad eventi, ed è proprio come lavora KStream in Kafka Streams. Ma se volessimo pensare a un concetto molto più vicino a una tabella di un database, dovremmo cambiare prospettiva. In uno stream di dati, ogni record può essere considerato indipendente dagli altri. Immaginiamo di avere dei record riguardanti acquisti di clienti. Supponiamo di voler effettuare il tracking delle attività di acquisti nel tempo. Se aggiungiamo una chiave che rappresenta l'ID del cliente, gli eventi di pagamento sono correlati l'uno

con l'altro, costruendo uno stream di aggiornamento, non di eventi. Se consideriamo questo stream di aggiornamento come un log, si ha a che fare con un changelog. Possiamo notare la differenza tra un log e un changelog: in un log, viene fatto l'append di ogni dato che arriva alla fine del file, mostrando alla fine tutti i record inseriti; in un changelog, vengono inseriti sempre i record in ingresso alla fine del file, ma in un changelog teniamo soltanto l'ultimo record inserito di una specifica chiave. È possibile forzare un log a mantenere solo l'ultimo valore aggiornato effettuando la compattazione.

Si può quindi pensare alla differenza tra uno stream di eventi e uno stream di aggiornamenti: consideriamo di avere come input un'azienda che comunica il prezzo attuale di mercato di tre prodotti per tre volte. In un KStream avremmo quindi nove record gestiti, ciascuno singolarmente, ma in una KTable questo non è vero: la KTable vede questi input come: tre input iniziali e due round di aggiornamenti degli input. Così alla fine la KTable avrà soltanto tre record, che sono i valori aggiornati finali dei prezzi. Si noti il fatto che in una KTable la chiave è fondamentale: senza quest'ultima non potremmo aggiornare un record in una tabella.

Una considerazione che dobbiamo fare è come fa la KTable a gestire i record e dove li memorizza. Al momento della creazione dell KTable, dovuta all'invocazione del metodo builder.table(..), lo StreamsBuidler genera in background uno StateStore per effettuare il tracking dello stato dello stream, creando così uno stream di aggiornamento. Lo StateStore creato in questo modo ha un nome interno non accessibile da query interattive. Esiste comunque una versione del metodo table che accetta un Materialized che specifica un nome di uno StateStore, che diventerà quindi accessibile.

Un'ulteriore considerazione deve essere fatta riguardo a quando e come KTable emette gli aggiornamenti ai processor successivi. Come premessa, è giusto dire che ci sono tre fattori discriminanti:

- Il numero dei record che scorrono nell'applicazione: più sono alti, più frequentemente verranno emessi gli aggiornamenti
- Quanti chiavi distinte ci sono nella tabella: più alto è il numero, maggiori saranno gli aggiornamenti.
- Configurazione di parametri: cache.max.bytes.buffering e commit.interval.ms.

Il fattore più rilevante è senza dubbio quello relativo alla configurazione dei parametri, in quanto l'unico che possiamo effettivamente impostare al momento dello sviluppo dell'applicazione. KTable si avvale del meccanismo di caching: la cache serve a deduplicare i record di aggiornamenti con la stessa chiave. Questo comporta una riduzione dei dati elaborati complessivamente, perché i nodi figli ricevono solo l'aggiornamento più recente e non tutti gli aggiornamenti. La dimensione massima di

caching è determinata dalla proprietà “cache.max.bytes.buffering”, che, se impostata a 0, fa sì che la cache non sia utilizzata in un’applicazione (attenzione, con cache disattivata si aumentano le operazioni di I/O, in quanto ogni singolo aggiornamento viene scritto).

L’altro parametro che abbiamo citato è “commit.interval.ms”, che serve per impostare quanto spesso, in millisecondi, è necessario effettuare il commit dello stato di un processor. Quando viene fatto il salvataggio (commit), viene fatto un flush della cache, mandando l’ultimo aggiornamento dei record ai processor successivi.

## 2.2.6.14 Aggregazione

Spesso, i dati in arrivo, se presi singolarmente, non danno troppe informazioni utili al contesto applicativo. Così, si rende necessario l’utilizzo di raggruppamenti e combinazioni di qualche tipologia. Per effettuare raggruppamenti sono presenti due metodi in Kafka Streams: KStream.groupBy e KStream.groupByKey. Entrambi restituiscono una rappresentazione intermedia, KGroupedStream, necessaria per effettuare successive operazioni di aggregazione. Infatti, effettuando un’aggregazione su questa rappresentazione, si ottiene sempre una KTable.

Esiste una differenza fondamentale tra i due metodi offerti: groupByKey è per quando un KStream ha già delle chiavi non nulle, facendo in modo che il flag booleano per il ripartizionamento (accennato quanche paragrafo fa) non venga modificato; groupBy assume, invece, che la chiave per il raggruppamento sia stata cambiata/modificata, quindi il flag viene impostato a “true”. Così, dopo aver chiamato groupBy, se effettuiamo un join, un’aggregazione e simili, il risultato sarà automaticamente ripartizionato. Si dovrebbe usare groupByKey quando possibile, piuttosto che groupBy.

```
KTable<String, ShareVolume> shareVolume = builder.stream(STOCK_TRANSACTIONS_TOPIC,
        Consumed.with(stringSerde, stockTransactionSerde)
                .withOffsetResetPolicy(EARLIEST))
        .mapValues(st -> ShareVolume.newBuilder(st).build())
        .groupBy((k, v) -> v.getSymbol(), Serialized.with(stringSerde, shareVolumeSerde))
        .reduce(ShareVolume::sum);


shareVolume.groupBy((k, v) -> KeyValue.pair(v.getIndustry(), v), Serialized.with(stringSerde, shareVolumeSerde))
        .aggregate(() -> fixedQueue,
                (k, v, agg) -> agg.add(v),
                (k, v, agg) -> agg.remove(v),
                Materialized.with(stringSerde, fixedSizePriorityQueueSerde))
        .mapValues(valueMapper)
        .toStream().peek((k, v) -> LOG.info("Stock volume by industry {} {}", k, v))
        .to("stock-volume-by-company", Produced.with(stringSerde, stringSerde));
```

Nell’esempio riportato, si presuppone di voler ispezionare la quantità di share delle aziende in un particolare settore, selezionando la prime cinque società più importanti in ciascun settore. Viene

creato uno stream che prende in input i dati delle società, successivamente viene fatto un map per ridurre la quantità di dati (vogliamo solo i dati sullo share, non tutti gli altri), utilizzando la lambda expression st -> ShareVolume.newBuilder(st).build(). Da qui, effettuiamo un groupBy, specificando la chiave con cui raggruppare per ogni azienda.Infine, si effettua una reduce, che è una forma di aggregazione ma che mantiene lo stesso tipo di oggetto (al contrario dell'aggregazione, che può anche restituire un tipo diverso di oggetto). In questo caso, la reduce viene fatta su un metodo di ShareVolume, il quale non fa altro che effettuare la somma degli share tra due record in ingresso. Adesso abbiamo lo share di ogni azienda in ogni settore. Per continuare, è necessario effettuare un altro raggruppamento, questa volta per settore, per poi fare un'aggregazione, punto cruciale in questo esempio. Invochiamo il metodo aggregate con primo parametro una lambda expression, che sarà in questo caso un mapping verso una coda a lunghezza prefissata (5) e con metodo di comparazione gli share delle aziende (non viene riportata qui la definizione di tale coda). Gli altri due parametri identificano il metodo per aggiungere un record e per eliminare un record. Ma analizziamo meglio l'aggregate. Una KTable, come è stato già detto, aggiorna record con la stessa chiave. Ovvero, sostituisce il valore vecchio con quello nuovo. L'aggregazione lavora proprio seguendo questa ideologia: aggrega i più recenti record con la stessa chiave. Se un record arriva, lo aggiunge alla coda (a lunghezza prefissata) usando il metodo adder, passato come secondo parametro. Ma se un altro record esiste già con quella chiave, è necessario sostituirlo e aggiornare il valore nuovo: per farlo, viene chiamato il metodo subtractor, terzo parametro, che rimuove il valore vecchio; successivamente, viene inserito il nuovo record aggiornato. Questo significa che la coda in questione non aggrega tutti i valori con la stessa chiave, ma terrà soltanto i primi N (5) che hanno il maggior share.

## 2.2.6.15 Windowing

In alcuni casi, vogliamo effettuare operazioni che coinvolgono record/eventi in un determinato intervallo di tempo, per esempio quante persone hanno ordinato un determinato prodotto nell'ultima ora. Kafka Streams mette a disposizione delle operazioni di windowing, suddivide in tre tipologie di funzionamento, accessibili mediante delle semplici invocazioni di metodo.

In generale, per poter effettuare operazioni con windowing, è necessario passare da uno stream ed effettuare un raggruppamento, per esempio groupBy, in modo tale da ottenere un'istanza di KGroupedStream. Da questa istanza, è possibile invocare il metodo windowedBy, il quale restituisce uno stream con windowing. A seconda del tipo di windowing scelto, ci può essere restituito o un

TimeWindowesKStream oppure un SessionWindowedKStream. Vediamo le tre tipologie di windowing esistenti:

- Session Window: molto differenti dalle altre due tipologie, non sono strettamente legate al concetto di tempo, ma a quello di attività degli utenti (o di qualunque altra entità che vogliamo). Infatti, la session window si basa sul concetto di sessione e sul concetto di inactivity gap.

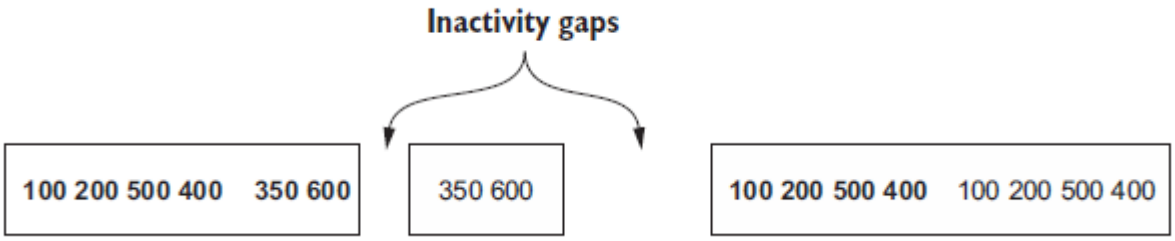


Figura 26. Inactivity Gap

In questo esempio possiamo vedere il funzionamento: la sessione piccola, contenente 350 e 600, sarà unita alla sessione di destra, perché l'inactivity gap è piccolo, mentre quella di destra rimarrà a sé stante, in quanto l'inacvtivity gap è grande.

```
windowedBy(SessionWindows.with(1000*30).until(1000*60*10))
```

In questo esempio, creaiamo una session window con inactivity gap di 30 secondi e un retention period di 10 minuti. Se avessimo adesso quattro record con la stessa chiave con ordine di arrivo sequenziale, per esempio 1=timestamp 12:10:10, 2=timestamp 12:10:20, 3=timestamp 12:10:55, 4=timestamp 12:10:00, questo significherebbe che all'arrivo del primo record, non ci sono sessioni aperte con record della stessa chiave, quindi viene creata la sessione con inizio e fine 12:10:10; arriva il record 2, si controlla che esista una sessione che finisca al massimo alle 12:09:50 oppure una che inizia al massimo alle 12:10:50. Viene trovata la sessione aperta dal record 1, quindi si fa il merge, mettendo il record 2 in questa sessione e ridefinendo i limiti, ovvero la sessione inizia alle 12:10:10 e finisce alle 12:10:20; arriva il record 3, controlliamo per una sessione per la stessa chiave ma non esiste, si crea una nuova che parte e finisce alle 12:10:55; arriva il record 4, si controlla per una sessione aperta per la stessa chiave tra le 12:09:30 e le 12:10:30. Vengono trovate le sessioni 1 e 2 (precedentemente unite), così tutti e tre i record vengono inseriti nella stessa sessione che parte alle 12:10:00 e finisce alle 12:10:20.

- Tumbling window: in questa tipologia di windowing, si catturano eventi entro un certo intervallo di tempo. Immaginiamo di dover catturare eventi ogni 30 secondi per un qualche

motivo, quindi li collezioniamo. Ogni 30 secondi viene creata una finestra di durata fissata, al cui termine cattura i dati. Si creano delle finestre distinte, senza sovrapposizioni e a durata fissa.

```
windowedBy(TimeWindows.of(1000*30))
```

- Sliding window: come la tumbling window, ma non attende che passi tutto il tempo prefissato prima di effettuare le operazioni necessarie, le fa periodicamente a intervalli più piccoli di tempo. Così vengono a crearsi finestre con sovrapposizioni ma che procedono a calcolare i risultati intermedi delle operazioni volute.

  ```
  windowedBy(TimeWindows.of(1000*30).advanceBy(1000*5)
  ```

  Nel caso in esempio, si crea una sliding window che avanza, cioè effettua il calcolo intermedio, ogni cinque secondi.

Ricapitolando, le session window non sono fisse nel tempo ma dipendono dall'attività degli utenti, le tumbling window danno un insieme di eventi entro un prefissato intervallo di tempo, le sliding window hanno una lunghezza fissa ma sono frequentemente aggiornate e possono contenere valori sovrapposti.

### 2.2.6.17 Join KStream e KTable

Potrebbe essere necessario aggiungere contenuti addizionali a degli stream in arrivo, magari prendendo i dati aggiuntivi da KTable. Vediamo come riusciamo ad effettuare un join tra un KStream a una KTable.

Il lookup nelle KTable è sempre effettuato sullo stato corrente della table, quindi dei dati in arrivo fuori ordine potrebbe scaturire in risultati non deterministici. Il risultato di un join tra un KStream e una KTable è un KStream (solitamente arricchito rispetto al primo). Per fare il join, è necessario assicurarsi che lo stream abbia una chiave. Se non la ha, bisogna prima di tutto estrarre una chiave con cui poter fare il join. Dopo averlo fatto, il join si presenta come quello che è stato introdotto negli appositi paragrafi, con l'esclusione dell'outer-join, non supportato in questa configurazione.

### 2.2.6.18 GlobalKTable

Nei paragrafi precedenti, si è introdotto e illustrato come aggiungere o arricchire i dati degli stream (di eventi). Si è affrontato il joining tra stream e tra stream e table. In ogni caso, nel momento in cui facevamo un mapping delle chiavi a un valore o tipo diverso, il ripartizionamento era necessario. La considerazione che deve essere fatta è che il ripartizionamento ha un costo. Si richiede di creare un topic intermedio, memorizzare i dati duplicati in un altro topic e si registra un aumento della latenza dovuta alla lettura e scrittura su un altro topic.

In alcuni casi, i dati con cui si vuole fare join sono relativamente piccoli e le intere copie di essi riescono a stare in un singolo nodo. Per questo tipo di situazioni, Kafka Streams offre le GlobalKTable. Con queste, i dati vengono replicati su ogni nodo. Poiché la totalità dei dati è quindi in ogni singolo nodo, non è necessario ripartizionare per chiave di lookup.

Il vantaggio di utilizzare una GlobalKTable sta proprio nell'avere una copia dei dati in ogni istanza/nodo. È possibile effettuare un join tra un KStream e una GlobalKTable con record senza chiave nello stream. Lo svantaggio principale è, ovviamente, la maggior occupazione di memoria dovuta alla replicazione dei dati. Pensiamo al join tra un KStream e una KTable: un join con record senza chiave e una table è possibile solo se prima di effettuare il join, estraiamo l'attributo di join e lo impostiamo come chiave (selectKey, map ecc). Nella pratica, ciò che facciamo è lo stesso procedimento di un join tra KStream e KTable, solo con i vantaggi e di non dover fare ripartizioni.

### 2.2.6.19 Processor API

Fino ad ora, si è discusso riguardo all'utilizzo delle API high-level di Kafka Streams: un DSL che dà la possibilità di creare applicazioni robuste ed efficienti con un minimo impiego di codice. Ma talvolta potremmo incorrere in problemi che richiedono di deviare rispetto all'approccio tradizionale. Questo impiegherebbe lo sviluppo di codice aggiuntivo, che si discosta dalle API che il DSL mette a disposizione e anzi è impossibile farlo, in quanto non modificabili.

Esistono molti trade-off tra un'astrazione high-level e un pieno controllo low-level. Un esempio tra tutti, il più significativo, è l'utilizzo dei framework ORM. Un framework ORM tende a mappare i tuoi oggetti di dominio in tabelle di un database, oltre che a creare query SQL runtime. Ma se ci dovessimo trovare di fronte ad una situazione in cui sono necessarie query molto complesse e intricate

tra loro, qualunque framework ORM non riesce a supportarle. In questi casi, è necessario scrivere codice SQL a low-level, per poter riuscire a ottenere le informazioni volute.

Le Processor API rispondono a questa problematica: permettono di effettuare operazioni (talvolta complicate) che il DSL non permette di fare. Per esempio, nelle KTable il framework controlla il timing per effettuare il forward dei record ai processor successivi. Se volessimo gestire noi questo tempo, dovremmo avere pieno controllo del framework. Con le Processor API ci riusciamo, con il DSL no.

Per costruire una topologia è necessario iniziare stabilendo quale nodo è il nodo sorgente. Per farlo, è possibile chiamare il metodo addSource della classe Topology. Nel metodo in questione, possiamo passare come parametri:

- Un nome del nodo nella topologia. Questo non veniva fatto nelle DSL API, in quanto l'istanza KStream generava automaticamente un nome. Il nome è utilizzato per cablare un nodo figlio a un nodo padre.
- TimestampExtractor. Il default è FailOnInvalidTimestamp, ma possiamo utilizzare altre tipologie (come descritto nel paragrafo sulla gestione dei timestamp).
- Key Deserializer e Value Deserializer. Nel DSL utilizzavamo istanze di Serde, le quali contenevano un serializzatore/deserializzatore: Kafka utilizza quello più appropriato, dipendentemente da se stiamo andando da un byte array a un oggetto o viceversa. Qua siamo in un contesto low-level, quindi dobbiamo dare direttamente i serializer/deserializer all'istanziazione.
- Nome del topic sorgente.

```
myTopology.addSource(EARLIEST, mySourceNodeName,
    new UsePreviousTimeOnInvalidTimestamp(),
    stringDeserializer, myDeserializer, Topics.MY_TOPIC.topicName());
```

Adesso cerchiamo di capire come aggiungere un nodo processor alla topologia. Per farlo, basterà chiamare il metodo addProcessor (anche utilizzando il fluent pattern, in quanto i metodi applicati all'oggetto Topology restituiscono la stessa istanza di Topology).

```
MyProcessor myProcessor = new MyProcessor();

myTopology.addSource(EARLIEST, mySourceNodeName,
    new UsePreviousTimeOnInvalidTimestamp(),
    stringDeserializer, myDeserializer, Topics.MY_TOPIC.topicName())
    .addProcessor(processorName, () -> myProcessor, mySourceNodeName);
```

Il metodo addProcessor prende come parametri:

- Nome del processor
- Il processor definito prima. Il secondo parametro sarebbe un'istanza di ProcessorSupplier, ma offre un solo metodo come interfaccia, quindi possiamo passare una lambda expression.
- Nome del nodo padre. Questo aiuta a stabilire una relazione tra nodo figlio e nodo padre. Al momento del build della topologia, il nodo figlio avrà un'etichetta che corrisponderà a chi è suo padre. Questo è essenziale per capire il flusso di movimento degli stream di un'applicazione.

Il processor deve o implementare direttamente l'interfaccia Processor (con tutti i metodi, init, process, punctuate e close) oppure estendere una classe astratta, chiamata AbstractProcessor, il cui unico metodo da implementare è process. Process è il punto chiave che viene chiamata quando un record attraversa la topologia.

Come ultima generalità, dobbiamo capire come aggiungere un nodo sink. È possibile farlo tramite il metodo addSink, che prende come parametri:

- Nome del sink.
- Topic che il sink rappresenta
- Serializer per la chiave
- Serializer per il valore
- Nodo padre per questo sink

Se aggiungiamo più nodi sink, per esempio due, con lo stesso nodo padre, allora abbiamo cablato entrambi i nodi sink al processor specificato.

Introduciamo come esempio una semplice applicazione per il word count e pensiamo a come poterla sviluppare con le Processor API. Prima di tutto definiamo un Processor custom, implementando l'interfaccia Processor e i relativi metodi.

```java
public class WordCountProcessor implements Processor<String, String> {

    private ProcessorContext context;
    private KeyValueStore<String, Long> kvStore;

    @Override
    @SuppressWarnings({ "unchecked", "deprecation" })
    public void init(ProcessorContext context) {
        this.context = context;
        kvStore = (KeyValueStore) context.getStateStore("counts");

        this.context.schedule(Duration.ofSeconds(1), PunctuationType.STREAM_TIME, (timestamp) -> {
            KeyValueIterator<String, Long> iter = this.kvStore.all();
            while (iter.hasNext()) {
                KeyValue<String, Long> entry = iter.next();
                context.forward(entry.key, entry.value.toString());
            }
            iter.close();
            context.commit();
        });
    }
    @Override
    public void close() {}

    @Override
    public void process(String key, String value) {
          String [] words = value.toLowerCase(Locale.getDefault()).split(" ");
          for (String word : words) {
              Long old = kvStore.get(word);
              if(old==null) {
                  kvStore.put(word, 1L);
              }
              else {
                  kvStore.put(word, old + 1L);
              }
          }
      }
}
```

Figura 27. Esempio Processor

Il metodo init è chiamato da Kafka Streams durante la fase di costruzione del task. In questo metodo vanno messe le operazioni di inizializzazione richieste al Processor per poter eseguire runtime correttamente. Il parametro di init è un ProcessorContext, il quale dà accesso ai metadati del record processato correntemente (Kafka topic, partizione, offset ecc). Una cosa molto importante è la possibiltà di effettuare uno scheduling dell'operazione di punctuation, tramite la chiamata context.schedule(), di effetturare il forward del record ai processor successivi, tramite context.forward() ed effettuare il commit del progresso corrente, tramite context.commit().

Nello specifico, context.schedule() accetta come parametri un Punctuator, che triggera un metodo punctuate() periodicamente, in funzione di PunctuationType. Il PunctuationType determina quale metrica di tempo/timestamp utilizzare nello scheduling. Esistono due tipologie:

- STREAM_TIME: punctuate è triggerato interamente dai dati, infatti il timestamp deriva dai record in input. Se no ci sono nuovi record in input, punctuate non viene chiamato. Esempio: effettuiamo scheduling ogni 10 secondi usando STREAM_TIME e processiamo 60 record con timestamp consecutivi da 1 a 60 secondi. Ciò che succede è che il metodo punctuate viene

innescato 6 volte, indipendentemente dal tempo richiesto per processare i record. Osservazione: questo tipo di gestione del tempo rende l'esecuzione di punctuate impredicibile: se non c'è flusso di nuovi dati, l'esecuzione non avviene.

- WALL_CLOCK_TIME: punctuate è triggerato solamente dal wall-clock-time, non dai record che fluiscono. Riutilizzando l'esempio appena citato, se i 60 record sono processati in 20 secondi, utilizzando però WALL_CLOCK_TIME, il metodo punctuate è chiamato 2 volte, cioè una volta ogni 10 secondi. Osservazione: l'utilizzo di questa opzione rende l'esecuzione di punctuate più predicibile.

In questo caso, nel metodo init scheduliamo la punctuation ogni secondo, oltre a configurare correttamente uno StateStore, che ci servirà per immagazzinare i risultati del conteggio. La punctuation schedulata non fa altro che iterare su tutti i valori disponibili nello store e farne il forward ai processor sottostanti.

Il metodo process è l'entry point delle operazioni da fare ogni volta che arriva un record in ingresso. In questo caso, controlliamo che l'input faccia o meno già parte dello Store. Se c'è già, incrementiamo il contatore del conteggio delle parole (parola già incontrata, quindi il conteggio aumenta); altrimenti, memorizziamo nello Store tale valore come nuova parola/entry.

Per creare uno StateStore, è sufficiente chiamare il metodo Stores.keyValueStoreBuilder, passando come primo parametro il tipo di Store (in-memory o persistent) e come altri parametri, i Serdes per la chiave e per il valore.

```
StoreBuilder<KeyValueStore<String, Long>> countStoreBuilder =
        Stores.keyValueStoreBuilder(Stores.persistentKeyValueStore("counts"),
            Serdes.String(), Serdes.Long());
```

In fine, è necessario creare la topologia connettendo i vari processor e lo state store, utilizzando l'oggetto Topology.

```
topology.addSource("Source", INPUT_COUNT_PROCESSOR)
    .addProcessor("Process", () -> new WordCountProcessor(), "Source")
    .addStateStore(countStoreBuilder, "Process") //Processor able to access this store = Process
    .addSink("Sink", OUTPUT_COUNT_PROCESSOR, "Process");
```

In questo caso, definiamo una sorgente, chiamata "Source", che si appoggia a un topic di input. Successivamente inseriamo un processor tramite addProcessor, specifichiamo il Processor custom che abbiamo creato precedentemente (Process) e il padre di questo processor (Source). Tramite addStateStore, agganciamo inoltre lo store appena creato a Process. Infine, aggiungiamo un nodo di sink, che ha come padre sempre Process.

## 2.2.6.20 Monitoraggio

Una parte cruciale delle applicazioni, anche al di fuori del mondo Kafka, è il monitoraggio. Verranno introdotte le principali tecniche di monitoraggio riguardanti i consumer e i producer Kafka, poiché Kafka Streams è una parte di Kafka, quindi monitorare Kafka Streams significa monitorare Kafka stesso.
Si possono considerare producer e consumer molto simili come infrastruttura, d'altronde uno produce i messaggi sul Kafka broker e l'altro li consuma: due facce della stessa medaglia. Per i producer, se parliamo di performance, possiamo intuire che ci preoccupiamo della velocità di produzione e invio dei messaggi al broker. Riguardo ai consumer, si ragiona in termini di velocità di lettura dei messaggi dal broker. Ma c'è un altro importante fattore da considerare: il consumer lag. Il consumer lag può essere definito come la differenza di velocità con cui i producer emettono messaggi sul broker e la velocità con cui i consumer li leggono. Come è possibile misurare tale consumer lag? Immaginiamo di avere un producer che emette dei messaggi, ogni messaggio avrà un certo offset. Il producer ha emesso 10 messaggi con offset da 400 a 410 e ha come ultimo messaggio inviato quello con offset 410. Ipotizziamo che un consumer stia leggendo tali messaggi e abbia come ultimo messaggio letto di cui ha fatto il commit il 405. Ecco che il consumer lag è in questo caso di 5 record (410 – 405).

Per effettuare il monitoraggio del consumer lag, Kafka mette a disposizione uno script, chiamato kafka-consumer-groups.sh, che si trova nella cartella bin dentro la directory di installazione di Kafka. L'opzione list che possiamo utilizzare con questo script permette di identificare tutti i consumer group attivi. Una volta che si esegue tale comando, è possibile selezionare il nome di un consumer group e utilizzare l'opzione describe, con cui si possono verificare il numero di messaggi letti, il numero di messaggi inviati al topic (prodotti dai producer) e il consumer lag. Ricordiamo che i consumer processano i messaggi in batch: finchè non hanno finito un batch, non iniziano a recuperare altri messaggi. Quindi un consumer lag non comporta necessariamente problemi.

Kafka offre la possibilità di intercettare sia i producer che i consumer per controllare e monitorare l'andamento dell'applicazione e capire quando e come intervenire per eventuali problemi.

Per l'intercettazione di consumer, sono presenti due access point per operare:

- ConsumerInterceptor.onConsume(): si interpone al record nel momento in cui il messaggio viene preso dal broker e appena prima che venga restituito al consumer tramite Consumer.poll(). Questo metodo accetta come parametro il record appena restituito dal broker e ha la possibilità di eseguire qualunque tipo di operazione su di esso, inclusi filtraggio o

modifiche, prima che il record sia restituito al consumer stesso tramite poll. È possibile impostare dei ConsumerInterceptor tramite ConsumerConfig.INTERCEPTOR_CLASSES_CONFIG con una collezione di uno o più classi che implementano ConsumerInterceptor. Se sono stati definiti più intercettori, essi eseguiranno in cascata nell'ordine con cui sono stati definiti nel ConsumerConfig. Se avviene un'eccezione durante l'esecuzione di più intercettori, la catena di intercezione non viene fermata: il record continua ad andare avanti nei prossimi intercettori (sebbene l'errore venga comunque inserito in un log).

- ConsumerInterceptor.onCommit(): dopo che il consumer effettua il commit dei suoi offset al broker, il broker restituisce una Map<TopicPartition, OffsetAndMetadata> che contiene informazioni sul topic, la partizione e gli offset di cui è stato fatto il commit, il tutto associato con dei metadati.

L'intercettazione dei producer è molto simile a quella dei consumer e lavora anch'essa tramite due access point:

- ProducerInterceptor.onSend(): l'intercettore può effettuare ogni tipo di azione, inclusa la modifica del record che sarà messo nel broker. Ogni intercettore nella catena riceve l'oggetto datogli in input dall'intercettore precedente. Stessa considerazione per le eccezioni.
- ProducerInterceptor.onAcknowledgement(): è chiamato quando il broker manda l'ack della ricezione del record. Se l'invio del record è fallito, viene comunque chiamato questo metodo.

Similarmente ai consumer, i ProducerInterceptor sono specificati in ProducerConfig.INTERCEPTOR_CLASSES_CONFIG, come una collezione di uno o più intercettori.
Osservazione: quando si configuarano gli intercettori in Kafka Streams, è necessario effettuare il prefix del consumer e producer interceptor , esempio di consumer: StreamsConfig.consumerProfix(ConsumerConfig.INTERCEPTOR_CLASSES_CONFIG).

## 2.2.7 KSQL

KSQL è un motore di streaming SQL per Kafka. Il suo utilizzo garantisce un abbassamento della barriera per il mondo dello stream processing, in quando offre una semplice e completamente interattiva interfaccia SQL per il processamento dei dati in Kafka: non è necessario scrivere codice con un linguaggio di programmazione. KSQL è distribuito, scalabile, affidabile e real-time. Supporta

un grande insieme di funzionalità di stream processing, come windowins, aggregazioni, join, sessionizzazione e molto altro.

KSQL offre nella pratica le stesse funzionalità di KStreams, solo in una sintassi SQL oriented: infatti, permette, per esempio, di creare uno stream di eventi, una table di changelog ed effettuare il join tra di essi. Quindi, ciò che KSQL fa è lanciare query continue, cioò trasformazioni che vengono eseguite continuamente quando nuovi dati fluiscono in stream nel topic.

Uno dei contesti di utilizzo in cui si pone KSQL è quello di real time monitoring: potrebbe essere utile monitorare il carico di CPU oppure definire delle specifiche condizioni di correttezza per un'applicazione e controllare real time se sono soddisfatte. Altri contesti possono essere quelli della sicurezza e rilevamento anomalie e integrazione dei dati.

```
CREATE TABLE possibile_frode AS
SELECT numero_carta, count(*)
FROM tentativi_di_autorizzazione
WINDOW TUMBLING (SIZE 5 SECONDS)
GROUP BY numero_carta
HAVING count(*) > 3;
```

Questo è un esempio per la creazione di una table contenente le carte che stanno subendo delle probabili frodi.

## 2.3 Spring

Spring è un framework a container leggero, utilizzato per semplificare il più possibile lo sviluppo di applicazioni Java Enterprise. I principali punti di forza di Spring sono l'aver introdotto l'AOP (Aspect Oriented Programming) come metodologia di sviluppo e il concetto di IoC (Inversion of Control).

Spring è modulare, cosa che consente di utilizzarlo sia nella sua interezza che solamente in parte, permettendo una facile integrazione anche con altri framework. Il modulo core è quello centrale e sempre necessario: garantisce le funzionalità fondamentali e fornisce Dependency Injection, AOP, gestione delle transazioni, applicazioni web, messaggistica, accesso ai dati, test e altro ancora. Altri moduli facoltativi possono essere inerenti a configurazioni di sicurezza, alle web app, ai Big Data, e molto altro.

## 2.3.1 Pattern MVC

Il pattern MVC è una risposta chiara all'intenzione di standardizzare e pulire l'architettura per le applicazioni web, in quanto mira a separare tutta la logica di business e la rappresentazione delle informazioni dalle interazioni utente con quest'ultime. Questo pattern è costituito da tre livelli:

- Model: implementa la logica di business tramite metodi utili per l'accesso ai dati dall'applicazione; gestisce eventuali DB.
- Controller: implementa la logica di controllo e riceve comandi dall'utente, solitamente dal livello "View", e li attua modificando lo stato degli altri due componenti.
- View: espone la parte di presentazione, intepretando i risultati ottenuti dal model e gestendo l'interazione con gli utenti. Sia Controller che View dipendono entrambi dal Model, il quale non dipende da nessuno: questo permette al Model di essere implementato e testato indipendentemente dagli altri livelli.

Grazie a questo pattern, è possibile implementare viste multiple, permettendo l'esposizione degli stessi dati allo stesso istante in modi diversi. Si riduce la complessità di sviluppo e il costo di aggiornamento, garantendo la manutenzione a uno dei livelli senza coinvolgerne altri.

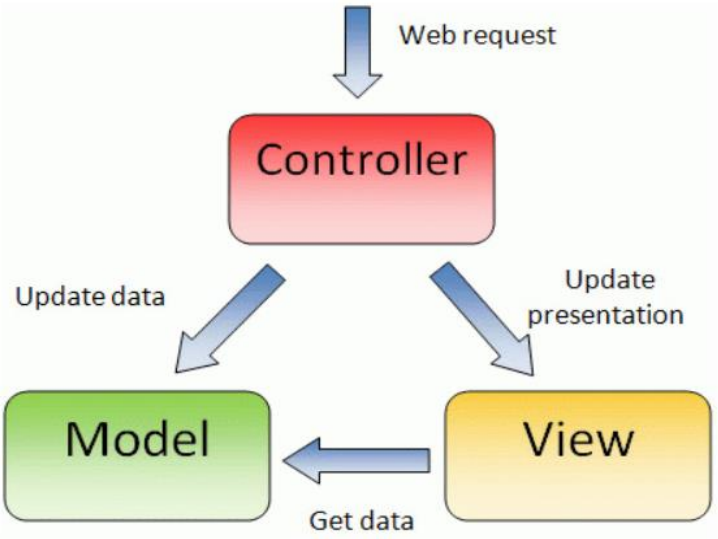


Figura 28. MVC di Spring

## 2.3.2 Spring Boot

Nato nel 2012, Spring Boot è una soluzione "convention over configuration" per il framework Spring del mondo Java, il quale mira a ridurre la complessità di configurazione di nuovi progetti Spring. Spring Boot definisce una configurazione di base che include le linee guida per l'uso del framework e tutte le librerie rilevanti di terze parti, facendo in modo di semplificare la creazione di applicazioni indipendenti e basate su Spring (proprio per questo, al giorno d'oggi molte applicazioni Spring sono basate anche su Spring Boot). Spring Boot offre diverse funzionalità vantaggiose, che lo hanno portato ad avere molto successo:

- Incorpora direttamente server o container nell'infrastruttura, senza dover quindi gestire a mano l'uso dei file WAR o EAR.
- Grazie all'introduzione dei POM (Project Objects Models), offre una configurazione semplificata di Maven.
- Porta una configurazione automatica di Spring, dove possibile.
- Fornisce anche caratteristiche non funzionali o configurazioni esternalizzate.

## 2.3.3 REST

Le API REST sono delle interfacce di programmazione delle applicazioni conformi a determinati vincoli imposti dall'architettura REST (REpresentational State Transfer), che consente interazione con servizi web RESTful. REST non è, quindi, un protocollo o uno standard, ma un insieme dei sopracitati vincoli architetturali. Quando una richiesta viene inviata tramite un'API RESTful, la risposta trasferisce uno stato rappresentativo della risorsa, solitamente consegnata tramite uno dei principali formati (JSON, HTML, XML ecc).

Molto importanti in ambiente RESTful sono anche le intestazioni e parametri nei metodi http, in quanto possono fornire informazioni di identificazione della richiesta, autorizzazioni, caching, cookie e molto altro.

Affinchè un'API sia considerata RESTful, deve rispettare alcuni criteri:

- Architettura client-server con richieste gestite tramite HTTP.
- Comunicazione client-server stateless, che quindi non prevede memorizzazione di informazioni del client; ogni richiesta è distinta e non connessa.

- Dati memorizzabili nella cache che ottimizzano le interazioni client-server.
- Un'interfaccia uniforme per i componenti, in modo che le informazioni vengano trasferite in forma standard. Quindi le risorse devono essere identificabili, possono essere manipolate dal client tramite la rappresentazione che ricevono (perché tale rappresentazione contiene le informazioni sufficienti alla manipolazione), le risposte contengono le informazioni necessarie per descrivere come il client deve elaborare l'informazione.
- Sistema su più livelli per ogni tipo di server che si occupa di recuperare le informazioni richieste in gerarchie.

### 2.3.4 Implementazione REST API con Spring Boot

Per riuscire ad implementare un'applicazione RESTful con Spring Boot, è sufficiente includere, durante la creazione del progetto, la dipendenza da "spring-boot-starter-web" (per esempio nel file pom.xml se stiamo usando Maven, altrimenti nel build.gradle se usiamo Gradle).

In un servizio web REST le risorse sono rappresentate da URI e le richieste http dirette verso questi URI permettono di interagire con le risorse esposte e di effettuare una serie di possibili operazioni. In Spring Boot è possibile gestire queste richieste tramite oggetti chiamati "Controller", che si occuperanno di effettuare un mapping tra i vari URI e le operazioni permesse a questi indirizzi.

Nel caso classico, come già detto, verrebbero utilizzati i "Controller", mentre nel caso RESTful, si dovranno utilizzare i "RestController", controller appositi per la gestione di risorse in servizi web REST.

Prima di procedere, una breve descrizione delle principali annotazioni di Spring Boot che possono essere utili in questo campo:

- @Controller: identifica un tipo di componente (@Component) che fa da controller per ricevere richieste web.
- @RequestMapping: associa un metodo del controller web a un'operazione HTTP (GET di default) per il path specificato; se applicato ad una classe, allora tutti i metodi hanno come path "root" il path indicato da tale annotazione.
- @ResponseBody: specifica che il valore restituito dal metodo va interpretato come contenuto della risposta; altrimenti, Spring interpreta il valore restituito come il nome della vista (View) da visualizzare al termine dell'esecuzione del metodo. Il valore di ritorno è automaticamente

serializzato in JSON sfruttando i campi dell'oggetto; è possibile specificare eventuali politiche in caso di campi non supportati.

- @RestController: è un @Controller con già tutti i metodi annotati con @ResponseBody, utilizzata per implementare web service REST.
- @GetMapping/PostMapping/ecc: specificano già che quel metodo sarà un handler per l'operazione di richiesta Get/Post/ecc per il path specificato come parametro.
- @RequestBody: simile a @ResponseBody, ma per la richiesta HTTP. Il corpo della richiesta viene automaticamente deserializzato in un oggetto Java a partire dall'oggetto JSON.

Per prima cosa, è necessario creare la Spring Boot Application, in modo da riuscire a far partire il motore che gestirà tutta l'applicazione. Per farlo, è sufficiente creare una classe come da figura X.

```
import org.springframework.boot.SpringApplication;

@SpringBootApplication
public class PacketDataMonitoringRestApp{

    public static void main(String[] args){
        SpringApplication.run(PacketDataMonitoringRestApp.class, args);
    }

}
```

Figura 29. Applicazione Spring Boot

Successivamente, possiamo implementare un RestController tramite l'annotazione "@RestController", come mostrato in figura 30.

```
@RestController
@RequestMapping("/myservice")
public class MyRestController{

    @Autowired
    private MyService service;

    @GetMapping
    public List<User> getAll(){
        return service.getListUsers();
    }

    @GetMapping("/{id}")
    public User getById(@PathVariable("id") long id){
        return service.getById(id);
    }

    @PostMapping
    public User create(@RequestBody User user){
        return service.create(user);
    }

}
```

Figura 30. Esempio RestController

Cerchiamo di capire bene come si possa implementare un RestController, sfruttando questo esempio: @RestController rappresenta una specializzazione di @Controller, che permette, in un colpo solo, di annotare tutti i metodi di handler con @ResponseBody; i metodi di MyRestController non restituiscono stringhe, bensì l'annotazione @ResponseBody permette di incorporare il valore di ritorno nel corpo della risposta HTTP in formato JSON; @RequestBody permette di effettuare il binding tra il corpo della richiesta e l'argomento del metodo

handler; il mapping delle richieste è coerente con le richieste del protocollo HTTP (GET, POST, ecc).

Effettuando una prova di get all'URI "host/myservice/1", verrà invocato il metodo getById con id=1. Se l'oggetto User ha i campi Nome, Cognome, Telefono e Mail, una possibile risposta, nel caso in cui l'utente con id 1 esista, potrebbe essere quella indicata in figura 31.


```
{
    "id": 1,
    "firstName": "Mario",
    "lastName": "Rossi",
    "phone": "3333333333",
    "email": "mario.rossi@gmail.com"
}
```


Figura 31. Esempio risposta JSON

È anche possibile utilizzare formati diversi dal JSON, per esempio XML. Per farlo, è necessario inserire come dipendenza il pacchetto che supporta la de/serializzazione di tale formato e specificare il media type corrispondente negli header delle richieste HTTP.

# 3. Progettazione, architettura e funzionalità di Dsrc Transaction Modules

Essendo lo scopo principale del progetto quello di garantire il processamento dei dati in ingresso per effettuare una serie di statistiche, la progettazione della soluzione si basa sul capire come impostare innanzitutto il back-end per il calcolo e mantenimento delle informazioni e, in secondo luogo, una metodologia per accedere facilmente ed efficientemente a tali informazioni. Il progetto, nel suo complesso, si chiama Dsrc Transaction Modules, suddiviso in sotto moduli, uno per ogni funzionalità.

## 3.1 Statistiche e Back-end

Inizialmente è stato necessario capire come impostare la parte relativa alle statistiche e back-end. I dati gestiti dall'applicazione del progetto saranno quelli provenienti da sensori autostradali, che possono essere pensati come un potenziale flusso continuo di dati. Ecco perché il nucleo centrale, come già anticipato, è Apache Kafka.

L'architettura del progetto, almeno per questa parte, si basa sulla definizione di una topologia Kafka Streams, a cui associare vari State Store e Topic di input e output. Per effettuare le statistiche, l'idea progettuale è quella di utilizzare maggiormanete gli State Store, piuttosto che i topic di output, utilizzando un retention time abbastanza lungo da permettere l'interrogazione con query temporali.

L'obiettivo principale è quello di calcolare due tipi di statistiche, relative a delle specifiche transazioni:

- Percentuali di transazioni fallite negli ultimi trenta giorni.
- Contatore di fallimenti consecutivi di transazioni istante per istante.

Nel caso di questo progetto, le transazioni saranno parsate da un parser, successivamente verranno estratti i dati rilevanti, dai quali si potranno poi ottenere delle informazioni riguardo al tipo di applicazione a cui fanno riferimento (per esempio, se sono Italiani o Spagnoli), all'OBU da cui provengono, allo stato della transazione ecc.

## 3.1.1 Topologia

Per realizzare l'applicazione è necessario avere ben chiaro in mente la topologia e il workflow che i dati andranno ad attraversare.

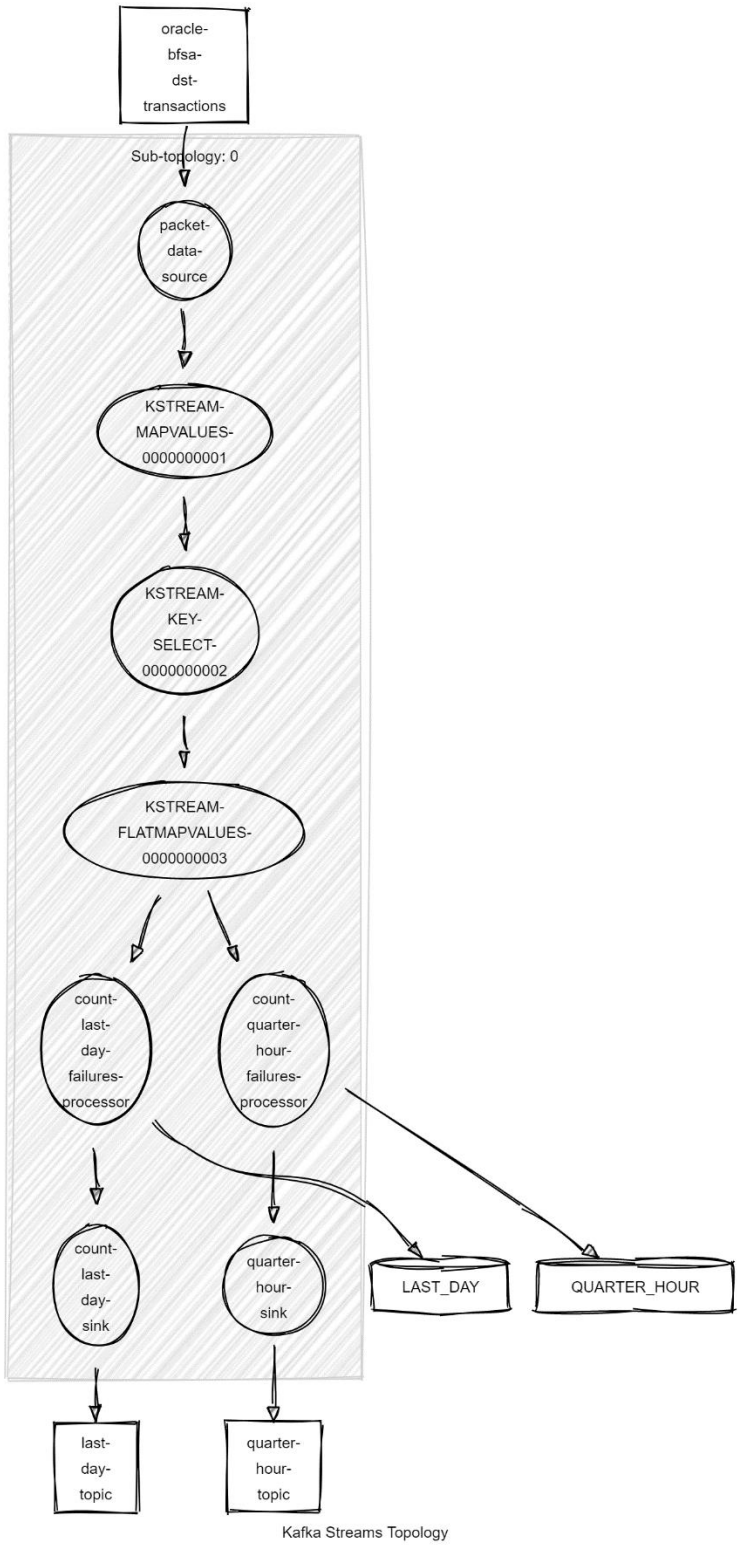


Figura 32: Topologia iniziale Progetto

La figura 32 riporta la topologia e i principali processor che si dovranno definire, oltre ai relativi State Store.

Prima di tutto, si definisce un topic di input, che farà da sorgente di tutti i dati che alimenteranno gli Stream all'interno dell'applicazione. Successivamente si avrà il primo processor di source, "packet-data-source", il quale è direttamente attaccato al topic di input. Poi si dovranno avere dei processor che effettuano operazioni di filtraggio per determinati tipi di transazioni che si vogliono monitorare, che si chiameranno "dsrc transaction" in questo progetto. I tre processor successivi al primo dovranno effettuare trasformazioni di questo genere, cercando di filtrare le informazioni necessarie e sufficienti per le statistiche che si vogliono calcolare, oltre a cambiare il valore della chiave e raggruppare i record come più sarà utile per i calcoli che si faranno in seguito. I due processor successivi, in cui si sdoppia lo stream principale, non sono altro che un modo per riuscire a raggruppare in finestre temporali distinte i dati elaborati. Infatti, durante la

progettazione della soluzione, è stato ritenuto utile effettuare questi tipi di operazioni, in quanto, al momento del calcolo delle statistiche, è possibile attuare delle query su finestre temporali, cosa che rende molto più riutilizzabile quest'applicazione per eventuali sviluppi futuri e altri calcoli di statistiche, senza essere legati solo ai ristretti requisiti iniziali. Stesso principio guida ha portato all'ideare due processor distinti con uno State Store ciascuno da cui poter prendere i dati per effettuare le statistiche; avere una granularità di finestre temporali da quindici minuti e da un giorno offre molte potenzialità al progetto. Tramite questa progettazione, è possibile riuscire a calcolare statistiche fino ad una unità base di quindici minuti, cosa che apre anche a statistiche di mezz'ora e così via. Invece, avendo anche inserito un processor (e quindi uno State Store) di un giorno, si riescono a fare statistiche su un qualunque numero di giorni, in particolare anche i trenta citati nei requisiti.

### 3.1.2 Dati

I dati provenienti dai sensori sono mappati in "oggetti" di diversi campi, alcuni dei quali binari/esadecimali. I dati che interessano questo progetto fanno parte di un campo binario che, una volta parsato correttamente, conterrà diversi campi:

- Numero di tratti che il dato contiene: infatti, un singolo OBU può avere eseguito transazioni su molteplici tratte, cosa che complica leggermente la gestione delle informazioni. I campi successivi sono da intendersi "per ogni tratto".
- Timestamp: istante di tempo in cui è avvenuta la transazione, utile per effettuare statistiche temporali.
- Id dell'applicazione (EID): per capire a quale applicazione fa riferimento la transazione; gli ID interessanti per questo progetto sono 1, 2, 3, 6, 8, 255.
- Campo di errore (ERR): a seconda dell'applicazione in cui siamo, può avere diversi valori, alcuni dei quali identificano errore.
    - EID=1 → errata se ERR diverso da 0xD3 e da 0x93
    - EID=2 → errata se ERR diverso da 0xD3 e da 0x93 e da 0x9B
    - EID=3 → errata se ERR diverso da 0xD3 e da 0x93
    - EID=6 → errata se ERR diverso da 0xD3 e da 0x93
    - EID=8 → errata se ERR diverso da 0xBB e da 0xDB e da 0xFB
    - EID=255 → errata se ERR diverso da 0xD3 e da 0x93
- Altri campi su informazioni che non riguardano questo progetto.

L’applicazione sarà impostata in modo tale da, una volta ricevuto il dato iniziale (quello in input al primo processor della topologia), parsare il campo binario in questione, estrarre da quest’ultimo i campi di interesse e andare a calcolare i dati per le statistiche, mettendoli nelle finestre temporali di ogni State Store.

## 3.2 RESTful API

Oltre la parte di calcolo statistiche e gestione dei dati, è necessario un modulo per poter accedere alle statistiche ed effettuare query temporali su di esse. Per fare questo, l’applicazione si avvarrà di Spring Boot e delle RESTful API che mette a disposizione, per poter creare un servizio ad-hoc per questo scopo.

Ovviamente, sarà anche necessario associare un output facilmente leggibile a ogni richiesta di statistiche da parte del modulo RESTful.

## 3.3 Estensioni

Una prima estensione dell’architettura si è resa necessaria dopo aver approfondito i requisiti richiesti. Infatti, non potendo utilizzare per questo progetto i Topic finali, in cui vengono immessi tutti i risultati delle statistiche, per la parte del modulo RESTful, l’unico metodo per accedere alle informazioni è quello degli State Store. Ma qua si apre un problema: gli State Store hanno un retention time, come spiegato nel capitolo precedente, e questo periodo non può essere troppo elevato, altrimenti viene meno proprio il concetto di “Store” e di “Topic”, in cui il primo serve a memorizzare per breve tempo, il secondo potenzialmente per sempre. L’analisi di tale problematica ha richiesto l’utilizzo di un altro tipo di soluzione: così come gli Stream in Kafka Streams hanno gli State Store, con retention time limitato, esistono i KTable, i quali si appoggiano ad un’altra tipologia di Store con una retention time simile a quella dei Topic, ossia potenzialmente illimitata. Sono quindi state introdotte, nella topologia, delle KTable che avessero il compito di memorizzare e permettere le query per i dati a lungo termine. Così, gli Stream, con gli State Store, rimangono per effettuare query su periodi corti, tipicamente qualche giorno, mentre le Table prendono vita per supportare le query su archivi storici di più lungo termine, tipicamente qualche settimana, ma anche mesi. Così, la Topologia iniziale si estende e

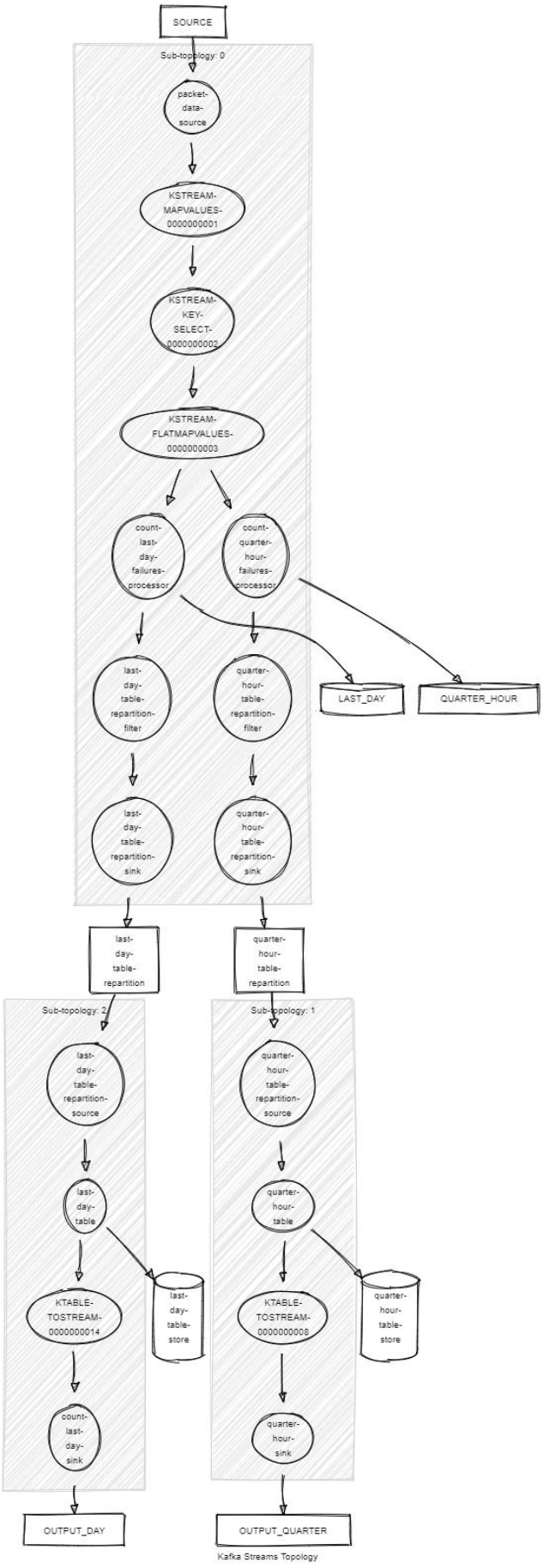


Figura 33. Topologia con prima estensione

diventa quella illustrata in figura 33. È possibile notare come la Topologia diventi suddivisa in due ulteriori sotto-topologie. Infatti, vengono sfruttati i raggruppamenti per giorni e per quarti d'ora per la creazione di due KTable, una per ciascun raggruppamento, che memorizzino negli appositi Store le informazioni a lungo termine. Viene quindi data la posisbilità di effettuare query con granularità di un quarto d'ora anche sull'archivio storico, così da avere maggior dinamicità. Ciascuna sotto-topologia emette i risultati finali in due Topic distinti.

Un'ulteriore estensione riguarda un concetto che cerca di stressare maggiormante lo streaming, essendo direttamente collegato al data rate dei record: anomaly detection. Sono state analizzate le varie configurazioni e possibilità che sarebbero potute esistere durante il ciclo di vita di questa applicazione. Effettivamente, la gestione dei dati in questione riguarda le transazioni ai caselli autostradali, con tutte le informazioni annesse (se è andata a buon fine, timestamp della transazione, ecc). Ecco che emerge la possibilità di riuscire a controllare il flusso di dati e analizzarlo per riconoscere dei pattern di anomalie in corso o successe. Tra tutte, quelle più rilevanti sono state due, di cui una scartata successivamente. La prima anomalia che si sarebbe potuta riscontrare è quella di un possibile furto/clonazione dell'OBU. Infatti, se in un arco di tempo basso le transazioni eseguite correttamente sono molte e in punti diversi, allora esiste un'elevata probabilità di furto/clonazione dell'unità. La seconda anomalia riguarda la possibilità di "fault" dell'OBU a seguito di tante transazioni fallite in una certa unità di tempo. Dopo aver analizzato approfonditamente gli scenari applicativi, è stato ritenuto praticamente impossibile la possibilità di clonazione che la prima strada non è stata considerata nella progettazione di questa estenzione. Così l'architettura è stata modificata solo per l'aggiunta di un'anomaly detection riferito alla seconda possibile anomalia. Osserviamo nella figura 34

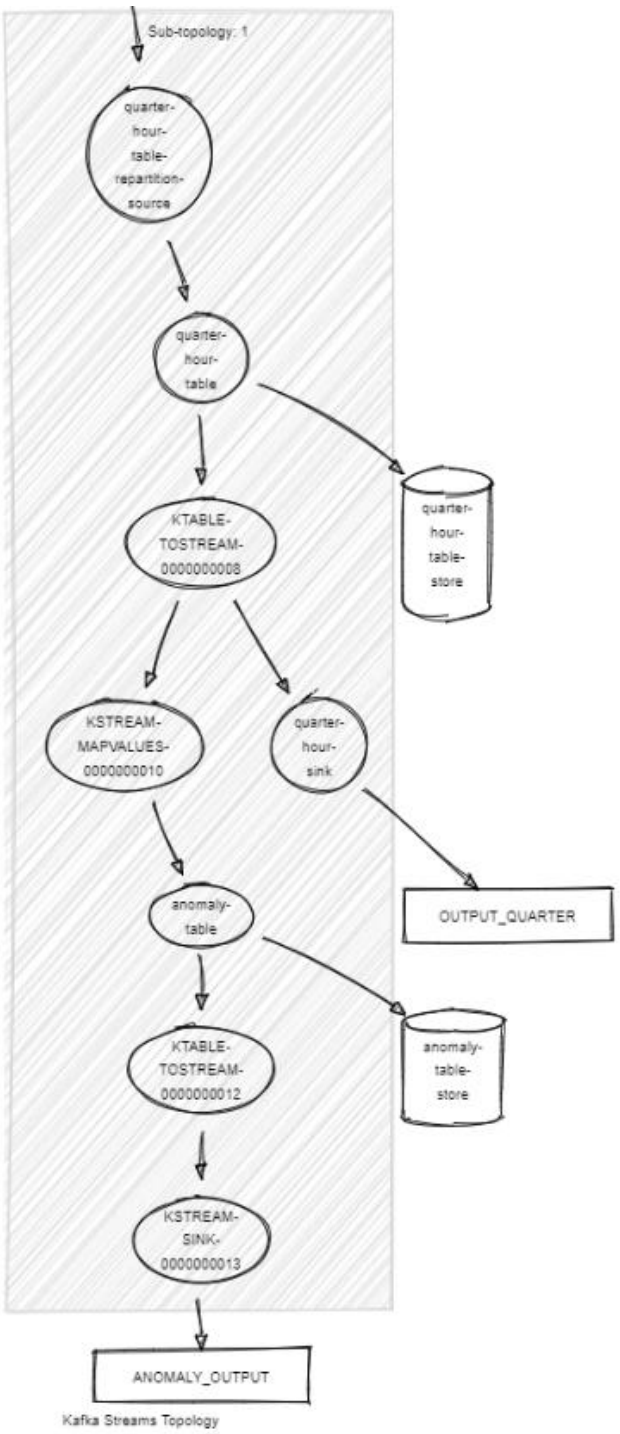


Figura 34. Topologia estensione anomalie

solo la parte cambiata dalla topologia mostrata in figura 33. È stato utilizzato il raggruppamento di quarti d'ora già presente, in quanto molto utile per intercettare anomalie di funzionamento degli OBU. Effettivamente, se ci fossero diverse transazioni fallite in un quarto d'ora da uno stesso OBU e EID, tale situazione potrebbe essere un "fault" dell'unità. È stata usata una KTable, per gli stessi scopi descritti nella prima estensione. Inoltre, si ha un topic finale, ANOMALY_OUTPUT, che serve per

la vera implementazione dell'anomaly detection: il Topic è alimentato da dati "filtrati", ovvero ci vengono inettate informazioni generiche riguardo a potenziali guasti/fault/anomalie nel sistema intero. La progettazione, infatti, è avvenuta cercando di rendere riutilizzabile il Topic, ovvero utilizzando un pattern generale che identifica per ogni OBU e EID, un eventuale messaggio e stato di anomalia che la descriva nel dettaglio, dando la possibilità di intervenire qualora fosse necessario. Il Topic è stato anche progettato in vista di potenziali Consumer che, collegati a tale Topic, restano in ascolto di nuovi record in arrivo. Qualora ci fossero nuovi record, allora questi identificano un potenziale fault e, quindi, un potenziale intervento umano. Ecco che i Consumer devono quindi notificare, in qualche modo, tali nuovi record.

# 4. Implementazione Dsrc Transaction Modules

In questo capitolo verrà affrontata l'implementazione della soluzione illustrata precedentemente. Verranno mostrate le principali tecnologie utilizzate per ogni fase del progetto. Infatti, durante la prima fase, è stato necessario collezionare un insieme di dati di test per poter riuscire a capire come effettuare il parsing corretto delle informazioni, utilizzando Docker Compose per la creazione di container locali, DBeaver per il collegamento a un database con dati di prova e Kafka Connect per creare un connettore che prendesse i dati (di test) dal DB e li iniettasse nel topic di input. Successivamente, è stato necessario implementare l'applicazione Kafka Streams, tramite l'utilizzo della topologia mostrata in precedenza, con tutta la logica applicativa. Infine, è stato sviluppato il modulo relativo all'applicazione Spring Boot RESTful, per accedere facilmente ai dati delle statistiche.

## 4.1 Gestione dati di prova

Come già citato, prima di iniziare a sviluppare la soluzione vera e propria è stato necessario riuscire ad accedere a dei dati di prova, per poter implementare un modo per effettuare il parsing e successivamente le operazioni per il calcolo delle statistiche.

### 4.1.1 Docker

Docker è un software open source per la containerizzazione, probabilmente il più noto a livello mondiale. La containerizzazione è un approccio nello sviluppo e rilascio del software basato su microservizi e su un metodo come il DevOps. È praticamente il contrario dell'approccio monolitic, che prevedeva una sola unità di deployment per un'intera applicazione: una piccola modifica comportava mettere mano all'intero codice del software.

Tramite un approccio granulare con i microservizi, ciascuno dei quali incapsulato in un container, è permesso agli sviluppatori di lavorare in contemporanea su una singola componente di una struttura più ampia, anche con tipologia di codice differente. Un altro punto a favore è la comunicazione tra i vari servizi: sono presenti delle API per riuscire a far comunicare i vari elementi tra di loro.

Docker può essere visto come un'evoluzione del paradigma SOA (Service Oriented Architecture), che rappresenta una scomposizione alternativa allo sviluppo monolitico. In particolare Docker si concentra sulla virtualizzazione come standard univoco di testing e deployment, ma non contiene le limitazioni tipiche delle VM, al contrario offre un ambiente leggero e portabile. La portabilità deriva dalla distribuzione basata su file immagine, ognuno dei quali ha deiversi layer, attivati a ogni modifica. Questa cosa comporta la possibilità di effettuare rollback e di ripristinare versioni precedenti, velocizzando i processi di deployment continui.

Docker Compose è uno strumento per lanciare applicazioni multi-container su Docker, definiti tramite un file in formato Compose. Il file è utilizzato per dettagliare come devono essere fatti i container dell'applicazione. Una volta creato il file, basterà lanciare Docker Compose tramite "docker-compose up".

Nella prima fase del progetto, è stato utilizzato Docker Compose per creare delle istanze locali di Kafka, per creare un connettore che si collegasse al DB di prova ed estraesse dei dati, così da poterli ispezionare e crearne un parser ad-hoc. Per fare questo, viene riportata la configurazione relativa solo al Kafka Connect, container rilevante per questa parte.

```
kafka-connect:
  image: confluentinc/cp-kafka-connect:latest
  depends_on:
    - zookeeper
    - kafka-c
    - schema-registry
  ports:
    - "8082:8082"
  restart: on-failure
  environment:
    CONNECT_ZOOKEEPER_CONNECT: 'zookeeper:2181'
    CONNECT_BOOTSTRAP_SERVERS: 'kafka-c:9092'
    CONNECT_REST_PORT: 8082
    CONNECT_GROUP_ID: connect-group
    CONNECT_CONFIG_STORAGE_REPLICATION_FACTOR: 1
    CONNECT_CONFIG_STORAGE_TOPIC: connect-config
    CONNECT_OFFSET_STORAGE_REPLICATION_FACTOR: 1
    CONNECT_OFFSET_STORAGE_TOPIC: connect-offsets
    CONNECT_STATUS_STORAGE_REPLICATION_FACTOR: 1
    CONNECT_STATUS_STORAGE_TOPIC: connect-status
    CONNECT_KEY_CONVERTER: org.apache.kafka.connect.storage.StringConverter
    #CONNECT_KEY_CONVERTER: org.apache.kafka.connect.json.JsonConverter
    #CONNECT_KEY_CONVERTER_SCHEMA_REGISTRY_URL: 'http://schema_registry:8081'
    CONNECT_VALUE_CONVERTER: org.apache.kafka.connect.json.JsonConverter
    #CONNECT_VALUE_CONVERTER_SCHEMA_REGISTRY_URL: 'http://schema_registry:8081'
    CONNECT_VALUE_CONVERTER_SCHEMAS_ENABLE: "false"
    CONNECT_INTERNAL_KEY_CONVERTER: org.apache.kafka.connect.json.JsonConverter
    CONNECT_INTERNAL_VALUE_CONVERTER: org.apache.kafka.connect.json.JsonConverter
    CONNECT_REST_ADVERTISED_HOST_NAME: kafka-connect
    CONNECT_PLUGIN_PATH: /usr/share/java
  networks:
    - packet-data-streaming-network
  volumes:
    - C:\Users\paverdini\DockerVolume:/usr/share/java/kafka-connect-jdbc/jars/
```

Figura 35. Configurazioni Docker Compose – Kafka Connect

## 4.1.2 Fetch dei dati

Per effettuare il fetch dei dati tramite Kafka Connect, è necessario creare un connettore che prenda come input i dati dal DB di prova e li metta in output da qualche parte. In questo caso, è stato sufficiente creare un Kafka Consumer attaccato al topic specificato dal connettore (vedere documentazione su Kafka) che mandasse in un file di testo i dati che leggeva. Successivamente, questo file di testo è stato usato per iniettare i dati di prova sul topic di input della topologia. In figura 36 si mostra il JSON per creare il connettore: tale file deve essere passato in una richiesta POST all'URL "localhost:8082/connectors", così da creare il connettore (ovviamente dopo aver attivato Docker Compose). Le informazioni riservate sono state oscurate o sostituite con stringhe casuali.

```
{
  "name": "dst_transaction_monitoring_test",
  "config": {
    "connector.class": "io.confluent.connect.jdbc.JdbcSourceConnector",
    "connection.url": "url",
    "connection.user": "user",
    "tasks.max":"1",
    "connection.password": "user",
    "db.timezone": "Europe/Rome",
    "dialect.name" : "OracleDatabaseDialect",
    "topic.prefix":"oracle-bfsa-dst-transactions",
    "mode": "bulk",
    "poll.interval.ms": 30000,
    "catalog.pattern": "PATTERN",
    "query":"SELECT * FROM ...",
    "timestamp.column.name": "COLUMN",
    "validate.non.null": false,
    "numeric.mapping": "best_fit",
    "transforms" : "createKey, extractString",
    "transforms.createKey.type":"org.apache.kafka.connect.transforms.ValueToKey",
    "transforms.createKey.fields":"C_OBU_ID",
    "transforms.extractString.type":"org.apache.kafka.connect.transforms.ExtractField$Key",
    "transforms.extractString.field":"C_OBU_ID",
    "key.converter":"org.apache.kafka.connect.storage.StringConverter"
  }
}
```

Figura 36. Configurazione Connector

Un'osservazione su questa configurazione può essere fatta sui campi "mode", "topic.prefix" e "query". Il primo se impostato a "bulk" identifica il fatto che i dati dal DB vanno presi tutti dall'inizio, compresi i dati già presenti nel DB. Infatti, riprendendo il discorso fatto sui Kafka Consumer nel capitolo 3, quando un Consumer viene creato e allacciato a un topic, inizierà a consumare i record arrivati da quel momento in poi, senza possibilità di prendere quelli precedenti. Poiché, in questo caso, si necessitavano di diversi dati per poterli analizzare e testare, la modalità "bulk" permette di accedere ai dati già presenti sul DB, senza aspettarne di nuovi. Il secondo identifica il topic in cui emettere l'output. Il terzo identifica la query con cui si interroga il DB; in questo caso, la query era una selezione della sola colonna con il campo binario da analizzare e parsare.

## 4.1.3 Parsing

I dati dal Topic di input arrivano serializzati in un oggetto Avro già esistente, che contiene tutte le informazioni di qualunque tipo all'interno delle soluzioni già sviluppate. Il primo passo è deserializzare tale oggetto e parsare le informazioni rilevanti solo al contesto di utilizzo attuale. Innanzitutto è stata implementata una classe che contenesse la lista dei tratti autostradali per ogni pacchetto di dati ricevuto. La classe è stata chiamata BfsDstInfo e contiene una lista di oggetti di classe DstTrack. Quest'ultima ingloba tutti i campi utili contenuti nel campo binario estratto precedentemente con il connettore.

```
public class BfsDstInfo {

    private List<DstTrack> tracks;

    private String C_OBU_ID; //Key

    public String getC_OBU_ID() {
        return C_OBU_ID;
    }

    public void setC_OBU_ID(String c_OBU_ID) {
        C_OBU_ID = c_OBU_ID;
    }



    public BfsDstInfo() {
        tracks = new ArrayList<DstTrack>();
    }

    public List<DstTrack> getTracks() {
        return tracks;
    }

    public void setTracks(List<DstTrack> tracks) {
        this.tracks = tracks;
    }

    public void addTrack(DstTrack track) {
        this.tracks.add(track);
    }

    @Override
    public String toString() {
        return "BfsDstInfo [C_OBU_ID=" + C_OBU_ID + ", tracks=" + tracks + "]";
    }
```

Figura 37. Classe BfsDstInfo

È stata implementata una classe che facesse il parsing dei dati in ingresso a partire dal pacchetto ricevuto. Il campo binario si chiama B_REQ.

```
public static BfsDstInfo getBfsDstInfo(Tbfa01ReqRsp packet) {

    try {
        UnifiedRequestBaseBean req = BfsRequestParser.parse(packet.getBREQ().array());
        return BfsDstParser.extractInfo(req, packet.getCOBUID());
    } catch (Exception e) {
        // return empty info, if error occurs while trying to extract information from packet
        return new BfsDstInfo();
    }

}
```

Figura 38. Parte di estrazione del campo binario B_REQ durante il parsing

Il metodo extractInfo imposta i valori dei campi interessati, estraendo le varie informazioni dal pacchetto in ingresso.

## 4.2 Applicazione Kafka Streams

Verrà qua analizzato lo sviluppo dell'applicazione Kafka Streams, quindi il cuore del progetto, seguendo la topologia già mostrata.

### 4.2.1 Topologia

Poiché è necessario suddividere per OBU tutti i risultati, dopo aver ottenuto le informazioni dal parser, è necessario inserirle in un oggetto Avro, creato appositamente, DstTrackInfoAvro, per poi selezionare come chiave degli Stream gli OBU di tali oggetti e come valore gli oggetti stessi. Infine, poiché ogni oggetto contiene una lista di transazioni/tratti, è necessario anche effettuare una flatMapValues, che mappa da una lista di transazioni a una transazione per record. Il processamento è quello mostrato in figura 39.

```
KStream<String, DstTrackInfoAvro> packetDataStream = PacketDataTopologyBuilder.createPacketDataStream(
        streamsBuilder, topics.get(FileProperties.Topic.SOURCE), new PacketDataTimestampExtractor(),
        schemaRegistryUrl)
.mapValues((k,v) -> BfsDstParser.getBfsDstInfo(v))
.selectKey((k,v) -> v.getC_OBU_ID())
.flatMapValues((k,v) -> v.getTracks());
```

Figura 39. Primi processamenti Topologia

Adesso è necessario effettuare tutte le operazioni su ogni record che portino a raggruppare per giorni e per quarti d'ora, calcolando allo stesso tempo le statistiche e le informazioni necessarie per i risultati finali. Sono state create due classi, che estendono la stessa interfaccia con unico metodo *define*, il quale aggiunge pezzi alla topologia, ciascuna delle quali ingloba la logica di un raggruppamento. In figura 40 si mostra come aggiungere tali raggruppamenti alla topologia originaria.

```
PacketDataTopologyBuilder.addWindowStores(streamsBuilder, stores, schemaRegistryUrl, usePersistentStore);
DSLTopology quarterHour = new CountEveryQuarterHourTopology(packetDataStream,
        topics.get(FileProperties.Topic.OUTPUT_QUARTER),
        topics.get(FileProperties.Topic.ANOMALY_OUTPUT),
        stores.get(FileProperties.Store.QUARTER_HOUR).getName(), schemaRegistryUrl);
quarterHour.define();

DSLTopology lastDay = new CountLastDayTopology(packetDataStream,
        topics.get(FileProperties.Topic.OUTPUT_DAY),
        stores.get(FileProperties.Store.LAST_DAY).getName(), schemaRegistryUrl);
lastDay.define();

Topology topology = streamsBuilder.build();
```

Figura 40. Raggruppamenti Topologia

Ovviamente, prima di inizializzare i vari componenti della topologia finale, è necessario creare i vari State Store con tutte le configurazioni del caso. Si istanziano poi le classi dei raggruppamenti, passando come parametri i Topic e Store di cui hanno bisogno. Da notare il fatto che il topic delle anomalie è stato passato solo al raggruppamento per quarti d'ora, questo per la motivazione spiegata nel capitolo precedente (anomalie con granularità "quarti d'ora" sono sufficienti). Infine, dopo aver invocato il metodo *define* di entrambe le classi, si effettua il build della topologia.

Adesso è il momento di analizzare i componenti che effettuano i raggruppamenti nel dettaglio.

### 4.2.2 Raggruppamenti per quarti d'ora

Analizzeremo qua il dettaglio del raggruppamento per quarti d'ora. Per prima cosa, si introduce il concetto che è stato utilizzato per raggruppare in finestre temporali, sia per i quarti d'ora che per i giorni. Ogni record deve essere inserito in una finestra temporale, potenzialmente insieme ad altri record che rientrano, come timestamp, in quella determinata finestra. Per esempio, se sono disponibili tre record, uno delle 13.05, uno delle 13.10 e uno delle 13.20 e stiamo raggruppando per quarti d'ora, allora esisterà una finestra temporale dalle 13.00 alle 13.15, in cui saranno messi i primi due record. L'ultimo record sarà inserito da solo nella finestra temporale che va dalle 13.15 alle 13.30. Ma la problematica maggiore è che, dopo aver inserito in una finestra temporale N transazioni, è necessario aggregarle tra di loro per calcolare le statistiche relative a quella finestra. Per esempio, se i primi due record sono entrambi dello stesso OBU e EID, il conteggio delle transazioni totali per quella finestra dovrà essere due, ovvero la somma dei conteggi totali. Inoltre, dobbiamo anche tenere conto delle transazioni fallite, quindi se uno dei due record fosse fallito, è necessario iniettarlo nel risultato finale, per poter avere le statistiche effettive per ogni finestra. Stessa cosa vale per il contatore delle transazioni fallite consecutivamente: dovranno essere calcolate, per ogni OBU e EID, le transazioni fallite consecutivamente e, se il totale finale di una finestra fosse diverso da zero, dobbiamo farlo visualizzare nel risultato finale. Questo porta all'introduzione di un ulteriore oggetto, creato sempre con Avro e chiamato AggregatePercentageFailedTransactions, che contiene i risultati per ogni finestra temporale, tra cui il totale di transazioni, quelle fallite, una percentuale delle fallite sul totale e il contatore dei fallimenti consecutivi. È stato poi creato un Serde Avro ad-hoc per questo nuovo oggetto, per poterlo serializzare e deserializzare negli Store e nei Topic appositi.

La soluzione implementata prevede l'utilizzo di un Transformer e un Aggregator. Il Transformer ingloba la logica di trasformazione dei record da <chiave obu, valore DstTrackInfoAvro> a dei record

del tipo <chiave TimeWindow, valore AggregatePercentageFailedTransactions>, dove TimeWindow identifica già una finestra temporale di un quarto d'ora. Prima di tutto, viene schedulata una punctuation ogni mezz'ora, per non sovraccaricare il processamento. La punctuation ha come effetto di verificare la presenza di finestra di quarti d'ora già completi e passati di cui farne il forward. Oltre a questo, colleziona i risultati parziali del quarto d'ora che sta passando e li imposta "da committare" per la successiva punctuation. Successivamente effettua il forward dei quarti d'ora conclusi.

```
@Override
public KeyValue<TimeWindow, AggregatePercentageFailedTransactions> transform(String key, DstTrackInfoAvro value) {

    // calcolo l'inizio del quarto d'ora di appartenenza utilizzando il timestamp della transazione
    TimeWindow timeWindowKey = TimeUtils.inferQuarterHour(value.getTimestamp());

    // prendo il record dallo state store
    ValueAndTimestamp<AggregatePercentageFailedTransactions> aggregate = this.store.fetch(timeWindowKey,
            timeWindowKey.start());

    if (aggregate == null) {
        aggregate = ValueAndTimestamp.make(new AggregatePercentageFailedTransactions(), timeWindowKey.start());
    }

    aggregator.apply(key, value, aggregate.value());

    this.store.put(timeWindowKey, aggregate, timeWindowKey.start());

    // non restituisco nulla nel transform, ma popolo solo lo store
    // la punctuate sarà quella che farà il commit
    return null;

}


private void schedulePunctuateCommitClosedQuarterHours(ProcessorContext context) {

    context.schedule(Duration.ofMinutes(30), PunctuationType.STREAM_TIME, timestamp -> {

        // recupera lo window store dal contesto
        TimestampedWindowStore<TimeWindow, AggregatePercentageFailedTransactions> store = getStore(context);

        Instant punctuateTime = Instant.ofEpochMilli(timestamp);

        SortedSet<Instant> uncommittedQuarterHours = uncommittedQuarterHours(store, punctuateTime);

        // prendi i  quarti d'ora vuoti da committare
        SortedSet<Instant> emptyQuarterHours = emptyQuarterHours(uncommittedQuarterHours, punctuateTime);

        uncommittedQuarterHours.forEach(QuarterHour -> forwardClosedQuarterHour(QuarterHour, store, context));

        emptyQuarterHours.forEach(QuarterHour -> forwardEmptyQuarterHour(QuarterHour, context));

    });

}
```

Figure 41 e 42. Parte del Transformer per raggruppamenti per quarti d'ora

Da notare il tipo di punctuation utilizzata, ovvero lo STREAM_TIME. Questo perché potrebbero esserci finestra temporali in cui non ci sono transazioni: usando il WALLCLOCK_TIME, si sprecherebbero delle punctuation a vuoto inutilmente.

L'operazione di transform si occupa invece di invocare l'Aggregator, che è quello che calcola effettivamente le statistiche, aggregando i vari record dei quarti d'ora. Il metodo "apply" è l'entry point che scatena l'aggregazione.

```
public class AggregatePercentageFailedTransactionsAggregator
    implements Aggregator<String, DstTrackInfoAvro, AggregatePercentageFailedTransactions>{

@Override
public AggregatePercentageFailedTransactions apply(String key, DstTrackInfoAvro value,
        AggregatePercentageFailedTransactions aggregate) {
    initializeCounters(aggregate);

    partitionId(taskId.partition, aggregate);

    count(aggregate);

    countMap(key, value.getEid() + "", aggregate.getCountTotalTransactions());
    if(failedTransaction(value)) {
        countMap(key, value.getEid() + "", aggregate.getCountFailedTransactions());
        increaseCounter(key, value.getEid()+"",aggregate.getCounter());
    } else {
        resetCounter(key, value.getEid() +"",aggregate.getCounter());
    }
    statistics(aggregate);
    return aggregate;
}
```

Figura 43. Metodo apply dell'aggregator

Dopo questi processamenti, viene invocato il metodo toTable per mettere nella KTable dei quarti d'ora i risultati ottenuti. Si rieffettua una trasformazione in KStream, tramite il toStream, per poi iniettare i record nel topic di output dei questi d'ora. Tutte queste operazioni devono essere eseguite specificano i vari serializzatori e deserializzatori Avro appositi, a seconda degli oggetti che stiamo memorizzando. In figura 44 è presente lo schema descritto finora.

```
var stream = packetDataStream
        // transform
        .transform(() -> new CountAndCommitQuarterHourTransformer(quarterHourStoreName),
                Named.as(TopologyElement.COUNT_QUARTER_HOUR.elementName()), quarterHourStoreName)
        // crea la KTable in cui saranno memorizzate a lungo termine le informazioni
        .toTable(Named.as(TopologyElement.TABLE_QUARTER_HOUR.elementName()),
                Materialized.<TimeWindow,AggregatePercentageFailedTransactions, KeyValueStore<Bytes,byte[]>>as
                (TopologyElement.TABLE_STORE_QUARTER_HOUR.elementName())
                    .withKeySerde(JsonSerdes.TimeWindow())
                    .withValueSerde(AvroSerdes.AggregatePercentageFailedTransactions(schemaRegistryUrl)))
        //Riporto a KStream
        .toStream();
        //Emetto su Topic di output dei quarti d'ora
stream.to(quarterHourAggrTopic,
                Produced.with(JsonSerdes.TimeWindow(),
                        AvroSerdes.AggregatePercentageFailedTransactions(schemaRegistryUrl))
                        .withName(TopologyElement.QUARTER_HOUR_AGGR_SINK.elementName()));
```

Figura 44. Implementazione Topologia raggruppamento quarti d'ora

## 4.2.3 Raggruppamento per giorni

Il concetto del raggruppamento per giorni è praticamente identico a quello relativo ai quarti d'ora, cambia l'arco temporale utilizzato per memorizzare i record e per invocare la punctuation. Nelle figure successive, si illustrano solo i principali punti rilevanti e di differenza rispetto ai quarti d'ora.

```
.transform(() -> new CountAndCommitLastDayTransformer(lastDayStoreName),
Named.as(TopologyElement.COUNT_LAST_DAY.elementName()), lastDayStoreName)
```

La puncutation del Transformer è schedulata ogni giorni, invece che ogni mezz'ora.

```
private void schedulePunctuateCommitClosedLastDay(ProcessorContext context) { //Schedule 1 day
    context.schedule(Duration.ofDays(1), PunctuationType.STREAM_TIME, timestamp -> {
        TimestampedWindowStore<TimeWindow, AggregatePercentageFailedTransactions> store = getStore(context);

        Instant punctuateTime = Instant.ofEpochMilli(timestamp);
        log.info("[{}] - Puncuation Time : [{}]", context.taskId(), punctuateTime.toString());
        SortedSet<Instant> uncommittedLastDays = uncommittedLastDays(store, punctuateTime);

        SortedSet<Instant> emptyLastDays = emptyLastDays(uncommittedLastDays, punctuateTime);

        uncommittedLastDays.forEach(lastDay -> forwardClosedLastDays(lastDay, store, context));

        emptyLastDays.forEach(lastDay -> forwardEmptyLastDays(lastDay, context));
    });

}
```

```
@Override
public KeyValue<TimeWindow, AggregatePercentageFailedTransactions> transform(String key, DstTrackInfoAvro value) {
    TimeWindow timeWindowKey = TimeUtils.inferLastDay(value.getTimestamp());
    ValueAndTimestamp<AggregatePercentageFailedTransactions> aggregate = this.lastDayStore.fetch(timeWindowKey,
            timeWindowKey.start());
    if (aggregate == null) {
        aggregate = ValueAndTimestamp.make(new AggregatePercentageFailedTransactions(),
                extractRightStartTimestamp(timeWindowKey.start()));
    }
    aggregator.apply(key, value, aggregate.value());

    this.lastDayStore.put(timeWindowKey, aggregate,  extractRightStartTimestamp(timeWindowKey.start()));


    return null;
}
```

Figure 45 e 46. Parte del Transformer per raggruppamenti per giorni

L'Aggregator è lo stesso utilizzato per i quarti d'ora, poichè le statistiche sono le stesse.

## 4.2.4 Gestione applicazione Kafka Streams

La gestione dell'applicazione Kafka Streams avviene tramite una classe PacketDataStreamManager, che contiene tutti i metodi di servizio, tipo per avviare/fermare l'applicazione oppure per effettuare le query sulle statistiche o anomalie.

```
public class PacketDataStreamManager {

    private static final Logger log = LoggerFactory.getLogger(PacketDataStreamManager.class);

    private static KafkaStreams kStreams = null;

    private Map<FileProperties.Topic, String> topics;
    private Map<FileProperties.Store, StoreInfo> stores;

    public boolean startProcessing(@NotNull String propertiesFileName) {

    public boolean startProcessing(@NotNull Properties properties) {

    public void endProcessing() {

    public State state() {

    public boolean isRunning() {
```

```
public WindowAnomaly getAnomalies(Instant instant, DsrcTransactionFilter filter){

public List<WindowAnomaly> getAnomalies(Instant from, Instant to, DsrcTransactionFilter filter){

public List<DsrcTransactionWindowStatistics> getLastDayStatistics(Instant from, Instant to, DsrcTransactionFilter filter){

public DsrcTransactionWindowStatistics getLastDayStatistics(Instant instant, DsrcTransactionFilter filter) {

public List<DsrcTransactionWindowStatistics> getLastDayStatisticsTable(Instant from, Instant to, DsrcTransactionFilter filter){

public DsrcTransactionWindowStatistics getLastDayStatisticsTable(Instant instant, DsrcTransactionFilter filter) {

public DsrcTransactionWindowStatistics getQuarterHourStatistics(Instant instant, DsrcTransactionFilter filter) {

public List<DsrcTransactionWindowStatistics> getQuarterHourStatisticsTable(Instant from, Instant to, DsrcTransactionFilter filter) {

//Per avere gli ultimi 30 giorni, basta passare Instant.now; in generale, per rendere tutto più flessibile,
//Il metodo utilizza un generico istante da cui calcolare a ritroso 30 giorni.
public DsrcTransactionWindowStatistics getLastMonth(Instant instant, DsrcTransactionFilter filter) {

public DsrcTransactionWindowStatistics getQuarterHourStatisticsTable(Instant instant, DsrcTransactionFilter filter) {

public List<DsrcTransactionWindowStatistics> getQuarterHourStatistics(Instant from, Instant to, DsrcTransactionFilter filter) {
```

Figura 47. Manager di tutti i servizi

Per facilitare il risultato dei metodi del manager, è stata prevista una classe DsrcTransactionWindowStatistics, che ingloba una finestra temporale e le statistiche calcolate relative a tale finestra e che espone tali informazioni tramite un JsonCreator, per uniformare le risposte verso il modulo RESTful.

```java
public class DsrcTransactionWindowStatistics {

    private static final Logger log = LoggerFactory.getLogger(DsrcTransactionWindowStatistics.class);

    private Instant windowStart;

    private Instant windowEnd;

    private Map<String, Map<String,Long>> totalTransactionsByObuAndByEid;

    private Map<String, Map<String,Long>> failedTransactionsByObuAndByEid;

    private Map<String, Map<String,Long>> counterConsecutiveFailedTransactions;

    private Map<String, Map<String,Float>> percentageFailedTransactions;

    @JsonCreator
    public DsrcTransactionWindowStatistics(@JsonProperty("windowStart") Instant windowStart,
            @JsonProperty("windowEnd") Instant windowEnd,
            @JsonProperty("totTransactions") Map<String, Map<String,Long>> totalTransactionsByObuAndByEid,
            @JsonProperty("failTransactions") Map<String, Map<String,Long>> failedTransactionsByObuAndByEid,
            @JsonProperty("counter") Map<String, Map<String,Long>> counterConsecutiveFailedTransactions,
            @JsonProperty("percentage") Map<String, Map<String,Float>> percentageFailedTransactions) {
        this.counterConsecutiveFailedTransactions=counterConsecutiveFailedTransactions;
        this.failedTransactionsByObuAndByEid=failedTransactionsByObuAndByEid;
        this.totalTransactionsByObuAndByEid=totalTransactionsByObuAndByEid;
        this.windowEnd=windowEnd;
        this.windowStart=windowStart;
        this.percentageFailedTransactions = percentageFailedTransactions;
    }
```

Figura 48. Classe DsrcTransactionWindowStatistics

Inoltre, possiamo vedere come ogni metodo accetti come parametro anche un DsrcTransactionFilter. Questo oggetto funge da filtro per OBU e/o per EID sulle query da effettuare ed è stato progettato appositamente per la correlazione con il modulo RESTful, da cui potranno essere richieste delle statistiche solo per determinati valori di OBU e EID. È stato utilizzato il pattern factory, da cui si utilizza un Builder per creare il filtro apposito con i valori da filtrare.

```java
public class DsrcTransactionFilter {

    private String obuId;

    private String eid;

    private DsrcTransactionFilter() {

    public static DsrcTransactionFilterBuilder Builder() {

    public static class DsrcTransactionFilterBuilder{

    public String obuId(){

    public String eid(){

    public String toString() {

}
```

Figura 49. Classe che implementa il filtro per OBU e EID

## 4.2.5 Estensione anomalie

La gestione delle anomalie, come già descritto nell'architettura, avviene solo per il raggruppamento per quarti d'ora. Quindi, la topologia esposta nel paragrafo dei quarti d'ora viene estesa con un'ulteriore processamento, che mappa tutti i dati in ingresso in potenziali record che possono essere potenziali anomalie. Per fare questo, è necessario implementare un ValueMapper, che dato un oggetto di statistiche in ingresso, restituisca un oggetto che contenga, per ogni OBU e per ogni EID, la presenza di eventuali anomalie. L'oggetto restituito è un Avro, implementato appositamente per essere poi memorizzato in Store di una KTable e nel Topic di output. In figura 50 vediamo il ValueMapper, che filtra per fallimenti consecutivi superiori a 5.

```java
public class AnomalyDetectionValueMapper implements ValueMapper<AggregatePercentageFailedTransactions,AnomalyDetectionAvro> {

    @Override
    public AnomalyDetectionAvro apply(AggregatePercentageFailedTransactions value) {
        AnomalyDetectionAvro result = new AnomalyDetectionAvro();
        var tot = new HashMap<String,Map<String,String>>();
        value.getCounter().forEach((obu,obuMap) -> {
            var map = new HashMap<String,String>();
            obuMap.forEach((eid,count) -> {
                if(count>5) {
                    map.put(eid, "Fallimenti consecutivi DSRC transaction troppo elevati, contatore=" + count);
                }
            });
            tot.put(obu, map);
        });
        result.setAnomaliesByObuAndEid(tot);
        return result;
    }

}
```

```
{
    "type": "record",
    "name": "AnomalyDetectionAvro",
    "namespace": "[redacted]packetdata.monitoring.streaming.c
    "doc":"Contiene possibili anomalie nel sistema - Messaggio di anc
    "fields": [
        {
            "name": "anomaliesByObuAndEid",
            "type": {
                "type": "map",
                "values" : {
                    "type": "map", "values":"string", "default":{}
                },
                "default": {}
            }
        }
    ]
```

Figure 50 e 51. Rispettivamente metodo apply del value mapper per filtrare le anomalie e schema dell'oggetto Avro utilizzato.

La topologia relativa alla gestione delle anomalie è quella rappresentata in figura 52. Prima viene fatto il mapValues, già spiegato, poi viene utilizzata una KTable per le query a lungo termine, con i

Serde relativi. Successivamente si riconverte il flusso di dati in KStream, tramite toStream, e lo si inietta nel topic di output.

```
stream.mapValues(new AnomalyDetectionValueMapper())
//Creo KTable per effettuare anche query da REST API, oltre che avere un consumer che emette segnali leggendo dal topic
.toTable(Named.as(TopologyElement.TABLE_ANOMALY.elementName()),
        Materialized.<TimeWindow,AnomalyDetectionAvro,KeyValueStore<Bytes,byte[]>>as
        (TopologyElement.TABLE_STORE_ANOMALY.elementName())
        .withKeySerde(JsonSerdes.TimeWindow())
        .withValueSerde(AvroSerdes.AnomalyDetectionAvro(schemaRegistryUrl)))
.toStream()
.to(anomalyDetectionTopic,
        Produced.with(JsonSerdes.TimeWindow(), AvroSerdes.AnomalyDetectionAvro(schemaRegistryUrl)));
```

Figura 52. Implementazione Topologia Anomalie

Come possiamo vedere in figura X (relativa allo stream manager), è stato anche creato un oggetto-risultato per la query sulle anomalie, chiamato WindowAnomaly, che contiene la finestra temporale in questione e i dati su eventuali record anomali, sempre tramite un JsonCreator.

```
public class WindowAnomaly {

    private Instant windowStart;

    private Instant windowEnd;

    private Map<String,Map<String,String>> anomalies = new HashMap<>();

    @JsonCreator
    public WindowAnomaly(@JsonProperty("windowStart") Instant windowStart,
            @JsonProperty("windowEnd") Instant windowEnd,
            @JsonProperty("anomalies") Map<String,Map<String,String>> anomalies) {
        this.anomalies=anomalies;
        this.windowEnd=windowEnd;
        this.windowStart=windowStart;
    }
```

Figura 53. Classe WindowAnomaly per incapsulare la risposta JSON

# 4.3 RESTful API

L'ultima parte riguarda l'implementazione del modulo Spring Boot per il servizio RESTful. Innanzitutto, l'applicazione è stata progettata per girare alla porta 8080 e gli sono state associate le configurazioni base tipiche di Spring Boot, tramite il file "application.properties". Verranno qua spiegate e descritte nel dettaglio solo le caratteristiche peculiari e più rilevanti di questo modulo.

### 4.3.1 Configurazione

È stato ritenuto cruciale essere il più possibili generici su tutti i vari parametri di configurazione sia dell'applicazione di Streaming che del modulo RESTful; per questo, è stato predisposto che tali parametri fossero passati da un file di configurazione esterno, denominato in questo caso "packetdata-monitdsrcapi-rest-app.properties". Qua possiamo trovare le proprietà riguardanti i vari Broker Kafka a cui connettersi, lo Schema Registry in cui inserire e da cui prelevare gli schemi Avro, le retention time degli State Store, il nome dei vari Topic e degli stessi State Store utilizzati nell'applicazione, oltre che tutte le altre principali configurazioni per Kafka Streams. In figura, un estratto di questo file.

packetdata-monitdsrcapi-rest-app.properties

```
### TOPIC

#Topic sorgente
packetdata.topic.source = oracle-bfsa-dst-transactions

#Topic output quarti d'ora
packetdata.topic.output.quarter.hour = quarter-hour-topic

#Topic output giorni
packetdata.topic.output.last.day = last-day-topic

#Topic output anomalie
packetdata.topic.anomaly.output = anomaly-topic

### STATE STORE

# Nome state store dei giorni
packetdata.store.lastDay = last-day-store

# Nome state store dei quarti d'ora
packetdata.store.quarterHour = quarter-hour-store
```

Figura 54. Configurazioni modulo REST

Per poter effettuare il caricamento delle proprietà da questo file, è stata predisposta una classe marcata con l'annotazione "@Configuration" in cui viene specificato dove trovarlo. Viene anche iniettata, in questa classe, l'implementazione del manager che si andrà ad utilizzare. Infatti, questo modulo RESTful è potenzialmente indipendente dal tipo di applicazione Kafka Streams sottostante, proprio perché tutte le dipendenze da esso sono state progettate in modo da iniettarle tramite file di configurazione. Questo rende tale modulo riutilizzabile e più dinamico.

```java
@Configuration
@PropertySource("file:${application.conf.dir}/packetdata-monitdsrcapi-rest-app.properties")
public class Config {

    @Value("${packetdata.stream.manager.impl}")
    private String packetDataStreamManagerImpl;

    @Bean
    public String propertiesFile() {
        return "packetdata-monitdsrcapi-rest-app.properties";
    }

    @Bean
    public PacketDataStreamManager packetDataStreamManager() throws Exception{
```

Figura 55. Classe Configuration per caricare le configurazioni

## 4.3.2 Servizi

L'implementazione della logica di business vera e propria è stata pensata nel modo più ingegneristico possibile: la riutilizzabilità e l'estensibilità sono i principi chiave di questo modulo. A tal fine, sono state create delle interfacce per ogni tipo di servizio, ciascuno dei quali espone delle operazioni specifiche. I servizi principali progettati sono uno relativo alle query interattive verso l'applicazione di Stream e un altro relativo a query sullo stato dello Stream (farlo partire, prendere lo stato attuale, stopparlo…). Di questi servizi si espongono le interfacce e le relative implementazioni con l'annotazione "@Service" di Spring Boot. Di seguito alcuni estratti.

```java
public interface InteractiveQueryService {

    public StreamResponse<List<DsrcTransactionWindowStatistics>> getQuarterHourStatistics
        (Instant from, Instant to,  DsrcTransactionFilter filter);

    public StreamResponse<DsrcTransactionWindowStatistics> getQuarterHourStatistics
        (Instant instant, DsrcTransactionFilter filter);
```

```java
public interface StreamService {

    /** Lancia l'istanza locale dell'applicazione stream caricando le properties da file.
    public boolean start(String propertiesFileName);

     * Effettua la chiusura dello stream per l'istanza locale.
    public void stop();

    /**
     * Restituisce true se l'istanza locale dell'applicazione kafka streams è in esecuzione;
     *
     * @return Se l'applicazione è in esecuzione.
     */
    public boolean isRunning();

     * Restituisce informazioni sullo stato di questa istanza locale.
    public StreamResponse<StreamStatus> streamStatus();


@Service
public class InteractiveQueryServiceImpl implements
    InteractiveQueryService, InitializingBean {

    @Autowired
    private PacketDataStreamManager packetDataStreamManager;

    @Autowired
    private StreamService streamService;

    public void afterPropertiesSet() throws Exception {

    @Override
    public StreamResponse<List<DsrcTransactionWindowStatistics>> getQuarterHourStatistics
        (Instant from, Instant to, DsrcTransactionFilter filter) {
        Supplier<List<DsrcTransactionWindowStatistics>> localQuery =
                () -> packetDataStreamManager.getQuarterHourStatistics(from, to, filter);

        Supplier<String> urlSupplier = () -> InteractiveQueryUtils
                .queryUrlWithParams(Endpoint.QUARTER_HOUR +
                        Endpoint.FROM_INTERVAL_STORE, from, to, filter);

        return getWindowStatisticsList(localQuery, urlSupplier);
    }
```

Figure 56, 57 e 58. Interfacce dei servizi offerti con esempio di implementazione e query su statistiche di un quarto d'ora

### 4.3.3 Wrapper Risultati

Per generalizzare i risultati ottenuti dall'applicazione di Stream, è stata introdotta una classe astratta wrapper generica che inglobi ogni tipologia di oggetto di risposta. Tale classe è StreamResponse.

```
public abstract class StreamResponse<T> {

    private final int totalHosts;

    private final int successHosts;

    private final T result;

    public StreamResponse(int totalHosts, int successHosts, T result) {

    public int getTotalHosts() {

    public int getSuccessHosts() {

    public T getResult() {

    public String toString() {

}
```

Figura 59. Wrapper per i risultati

Per ogni tipo di oggetto di cui si vuole incapsulare il risultato, è sufficiente estendere StreamResponse parametrizzata con questo oggetto e crearsi un costruttore ad-hoc, con un JsonCreator per il renderig della risposta. Per esempio, nel caso delle statistiche per questo progetto, è bastato creare una classe DsrcTransactionWindowStatisticsResponse che estendesse StreamResponse con l'oggetto DsrcTransactionWindowStatistics.

```
public class DsrcTransactionWindowStatisticsResponse extends
    StreamResponse<DsrcTransactionWindowStatistics>{

    @JsonCreator
    public DsrcTransactionWindowStatisticsResponse(
            @JsonProperty("totalHosts")int totalHosts,
            @JsonProperty("successHosts") int successHosts,
             @JsonProperty("result") DsrcTransactionWindowStatistics result) {
        super(totalHosts, successHosts, result);
    }
```

Figura 60. Classe DsrcTransactionWindowStatisticsResponse per incapsulare le risposte

Allo stesso modo, per le anomalie è stata creata una classe che wrappasse l'oggetto WindowAnomaly, come citato precedentemente. Per risultati come lista di statistiche, è stato sufficiente creare un wrapper di una lista di oggetti.

### 4.3.4 Rest Controller

Le componenti cruciali per il funzionamento della REST APP sono i Controller, implementati utilizzando l'annotazione @RestController. Per ciascuna funzionalità di query è stato associato un RestController con un certo endpoint di mapping e determinati metodi in risposta a richieste HTTP in ingresso.

```
@RestController
@RequestMapping(Endpoint.DAY)
public class PacketDataMonitdsrcapiLastDayRestController {
    private static final Logger log = LoggerFactory.getLogger(PacketDataMonitdsrcapi

    @Autowired
    private StreamService streamService;

    @Autowired
    private InteractiveQueryService interactiveQueryService;

    @GetMapping(Endpoint.FROM_INTERVAL_STORE)
    public Object fromIntervalStore(@NotNull @RequestParam("from") Long from,
            @NotNull @RequestParam("to") Long to,
            @Nullable @RequestParam("eid") String eid,
            @Nullable @RequestParam("obu") String obu) {
```

Figura 61. Controller per richieste su giorni ed esempio di metodo query

Per ciascuna funzionalità, quindi query su giorni, query so quarti d'ora, query sugli ultimi trenta giorni e query sulle anomalie, esiste un RestController con appositi metodi. I metodi previsti sono differenziati su dove vogliamo effettuare la ricerca delle statistiche, per esempio se sugli State Store per intervalli di tempo non troppo lontani, oppure sulle KTable per query a lungo termine. Inoltre, è possibile effettuare, su ognuno dei RestController, interrogazioni su istanti di tempo, da cui viene inferito l'unità di misura del servizio in cui siamo (per esempio, se l'istante di tempo specificato fosse dentro il RestController dei giorni e identifica il 20 agosto 2020 alle ore 10.05 UTC, allora viene inferito che si vogliono le statistiche di quel giorno preciso), ma anche interrogazioni su intervalli temporali qualunque (per esempio, stessa situazione di prima ma con query dal 20 agosto al 25 agosto). Da notare il fatto che nell'URL di richiesta HTTP possono anche essere specificati OBU e EID, opzionalmente; se presenti, attivano il filtraggio dei risultati solo su quei valori specificati, tramite il filtro DsrcTransactionFilter.

```
DsrcTransactionFilter filter = DsrcTransactionFilter.Builder()
        .eid(eid)
        .obuId(obu)
        .build();
```

Figura 62. Building del filtro

## 4.4 Test e Performance

La misurazione di performance è avvenuta sia nell'ambiente di prova locale, che dopo aver pubblicato e messo in ambiente di esecuzione le applicazioni. In entrambi i casi, le performance delle soluzioni adottate prima non sono neanche paragonabili con quelle attuali: effettuando precedentemente delle query su DB per avere questi dati, si passa da un ordine di grandezza di molti secondi (sforando anche il minuto), fino a parlare di millisecondi, o comunque centinaia di millisecondi.

### 4.4.1 Test Locali Junit

I primi test di performance sono stati effettuati in ambiente locale, come già accennato. I test sono stati fatti per ogni tipologia di query possibile e su ogni raggruppamento ed estensione. La prima parte di questi test locali comporta l'utilizzo di Junit e Kafka Streams per test unitari, in cui le varie istanze (Broker, Zookeeper ecc) sono simulate da Junit stesso. Questo è stato fatto per calcolare le varie tempistiche di accesso agli Store, sia gli State Store dei KStream, che alle KTable. Sono stati simulati prima 200 record in ingresso al topic di input, successivamente 20000 record, per vedere il grado di scalabilità e performance complessiva nell'accesso agli store. In questo caso, le prove sono state fatte sui raggruppamenti per giorni e sull'estensione delle anomalie.

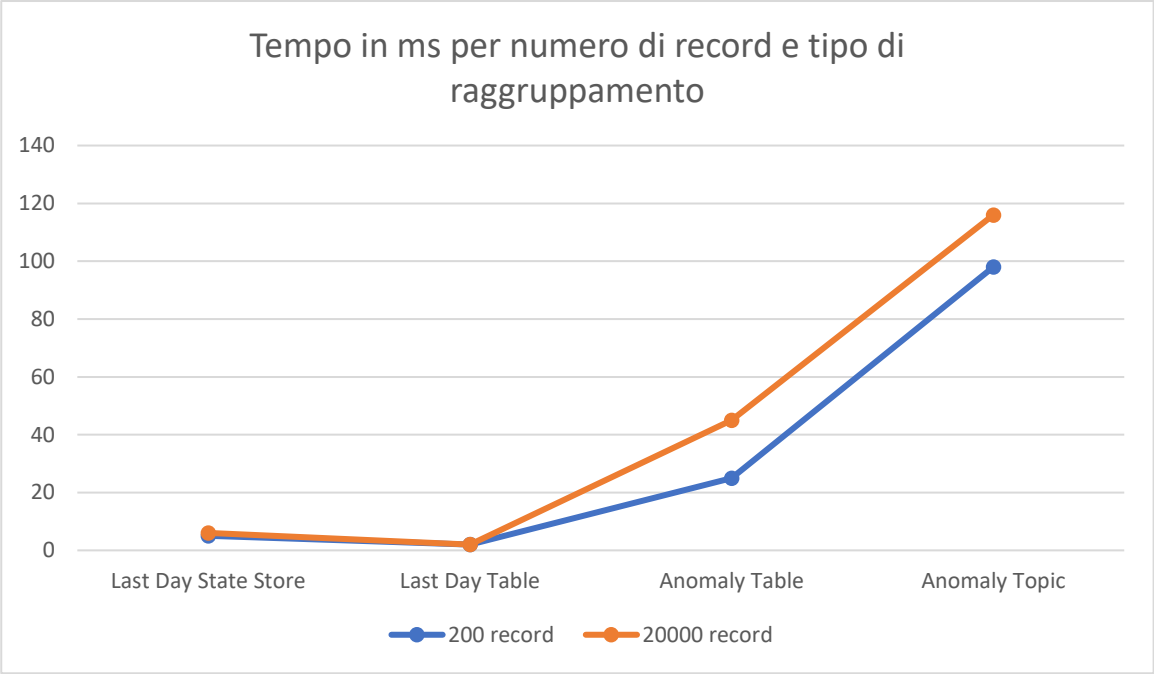


Figura 63. Test giorni e anomalie in locale con Junit

L'asse delle ordinate rappresenta il tempo totale di fetch dei dati in millisecondi. Possiamo notare come l'accesso ai vari Store per il raggruppamento in giorni non cambia quasi per niente al variare del numero di record, cambia invece leggermente di più per quanto riguarda l'estensione anomalie. Comunque, il tempo maggiore calcolato per accedere a questi dati è di 116 ms e riguarda l'anomaly topic, un valore comunque molto contenuto se paragonato alle soluzioni antecedenti questo progetto di tesi.

### 4.4.2 Test Locali Docker

Per attuare queste prove, è stata predisposta un'istanza locale di Kafka tramite Docker Compose, che integrasse anche il Control Center di Confluent. Il Control Center permette di accedere facilmente a ogni tipo di informazione, consumo, overhead, topic, consumer lag, performance e molto altro per ciascun cluster di Broker implementato, cosa che si presta molto bene alla rilevazione delle prestazioni per i fini del progetto.

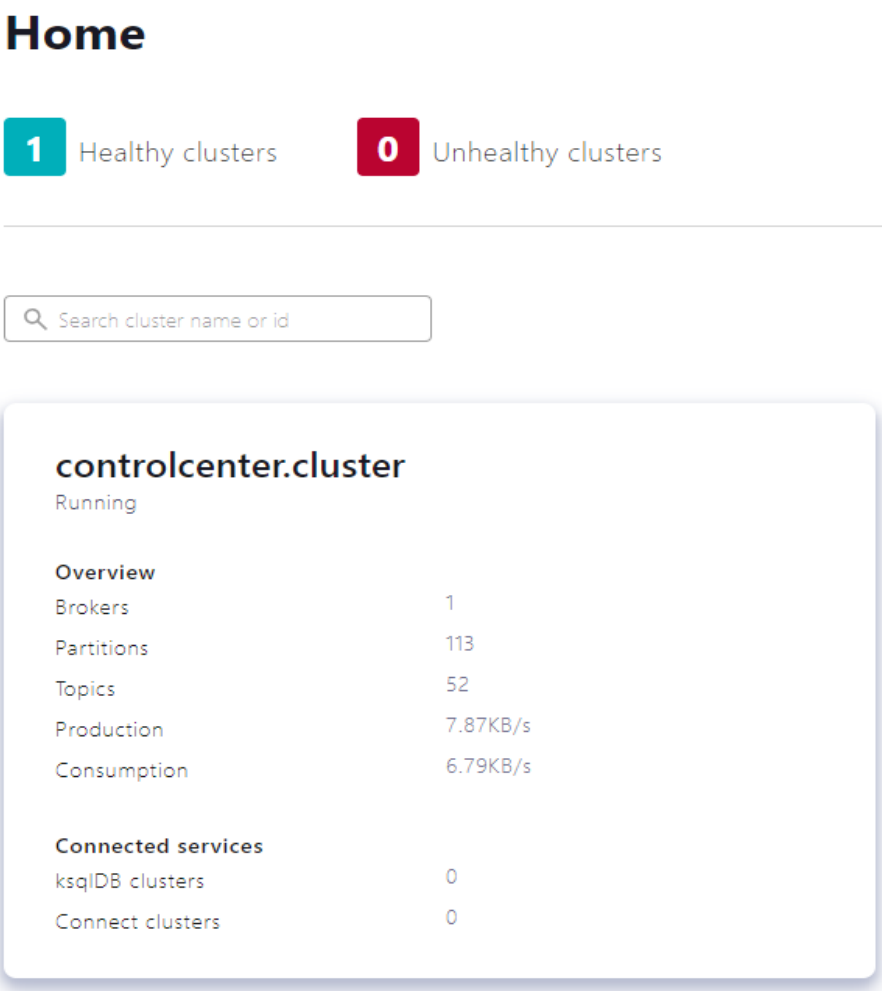


Figura 64. Interfaccia Control Center

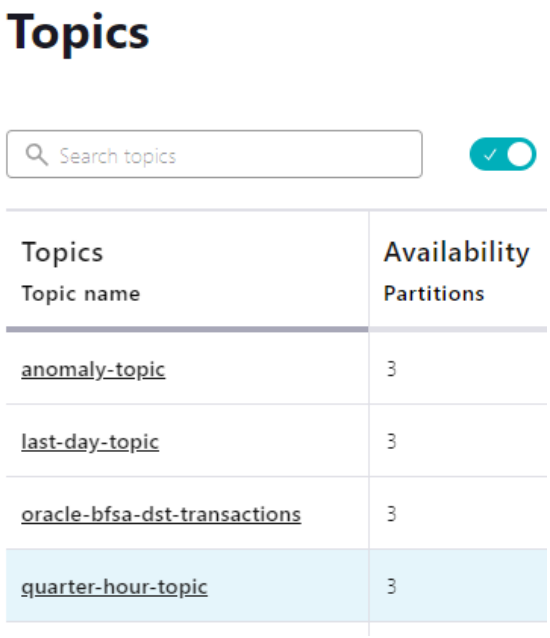


Figura 65. Sezione “Topics” del Control Center

I test effettuati in ambiente locale con Docker comportano l'utilizzo del modulo REST, tramite il quale è possibile effettuare ogni tipo di query sia sulle statistiche che sulle anomalie. Di seguito si riportano le tempistiche in millisecondi riguardanti alcune chiamate di prova.

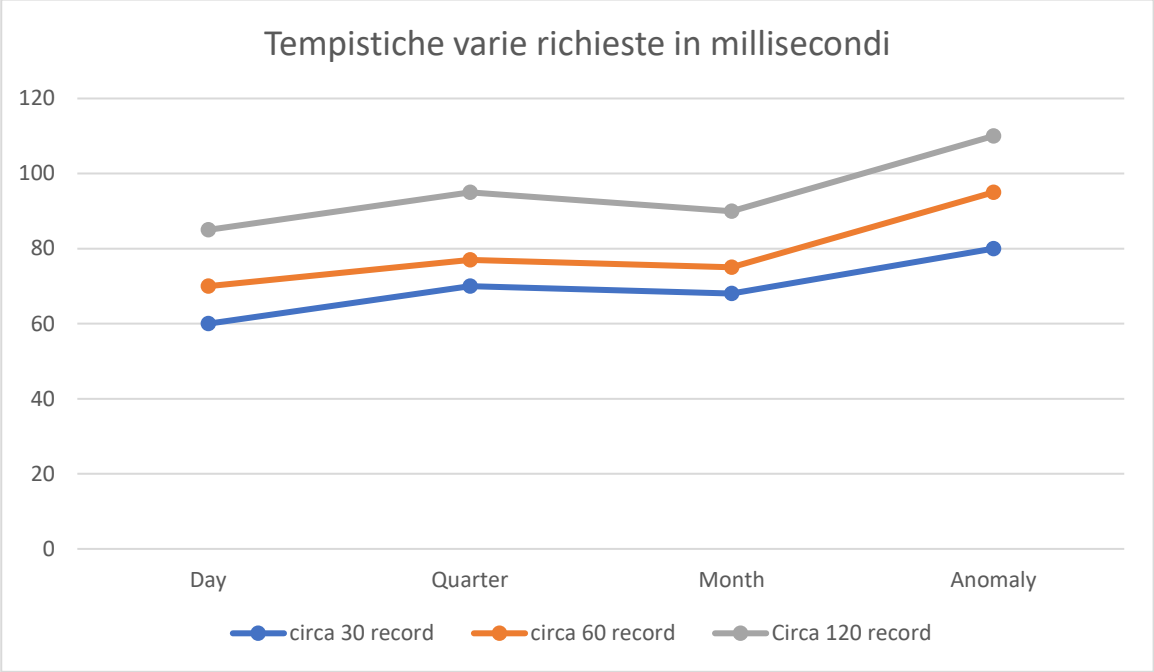


Figura 66. Tempistiche varia chiamate di prova

Per misurare queste prestazioni, sono stati simulati circa 30 record, poi 60 e infine 120 per ogni tipologie di query possibile, ovvero per giorni, per quarti d'ora, per mese e per le anomalie. Le tempistiche di accesso sono notevolmente ridotte, soprattutto comparando con le vecchie metodologie utilizzate (come già detto, query SQL). Infatti, a scopo di paragone, per l'effettuazione di una query per OBU in un certo periodo con qualche decina di record come risultato, sono necessari circa 15 secondi. Inoltre, la query per OBU utilizzando SQL direttamente nel database non permette il real time processing e nemmeno il filtraggio per EID: quest'ultimo, infatti, è racchiuso in una colonna dei risultati che contiene un campo binario, precedentemente descritto (B_REQ).

Adesso analizziamo, invece, le performance del topic delle anomalie, quello più interessante in quanto accessibile direttamente per monitorare le anomalie stesse. Il test è stato fatto implementando un producer di anomalie e un consumer delle stesse, in modo da testare il numero di produzioni e di consumi, per poi utilizzare il Control Center come entry point delle informazioni sulle performance. Sono stati simulate 500 anomalie, con altrettante letture dal consumer.

| Consumer Lag Massimo | Production | Consumption | Availability |
|---|---|---|---|
| 2 | 10 KB/s | 8 KB/s | Sempre |

Dai test di performance del topic delle anomalie, possiamo notare subito quanto non ci siano problemi: effettivamente, un consumer lag massimo di 2 è insignificante, in quanto può anche essere dovuto al batching dei record da parte del consumer. I dati sulla produzione e sul consumo sono standard e non comportano ulteriori analisi. La disponibilità non è mai venuta a mancare, complice anche il fatto di essere con istanze locali e senza particolari vincoli.

### 4.4.3 Test Ambiente di rilascio

Purtroppo, l'applicazione non è stata ancora rilasciata al momento della conclusione di questa tesi, pertanto non sono stati effettuati test delle prestazioni sull'ambiente di rilascio. Nonostante ciò, appena il progetto sarà messo in esecuzione, il contenuto di questo paragrafo sarà aggiornato con i dati risultanti dal test delle performance.

# Conclusioni

L'incremento dei dispositivi che emettono continuamente informazioni è sempre in rapido aumento. La tecnologia deve stare al passo, al fine di riuscire a processare, collezionare e monitorare i dati in arrivo da questi dispositivi. In effetti, esistono moltissime tecnologie in grado di risolvere e ottimizzare i requisiti dei Big Data, sia in ambito di Batch Processing che di Stream Processing.

Storm, Spark e Flink sono solo alcune delle piattaforme di Stream Processing divenute famose e ampiamente utilizzate attualmente. L'approccio con cui risolvono le problematiche dello Stream Processing è diverso da soluzione a soluzione, ma le feature che devono offrire tali piattaforme sono comuni: a prescindere da ogni tipo di accadimento, un particolare record in arrivo alla piattaforma di Stream Processing deve essere processato, in caso di fallimenti, la piattaforma deve essere in grado di ripristinare l'ultimo stato consistente e da lì ripartire, possibilità di mantenere stato in caso di operazioni stateful, latenza più bassa possibile e throughput più alto possibile.

Oltre alle feature sopra elencate, è necessario capire che, al fine di scegliere una piattaforma piuttosto che un'altra, ci sono altri fattori, quali la possibilità di feature avanzate (Event Time Processing, Watermarks e Windowing) e la maturità (se la piattaforma è già stata provata e la scalabilità è già stata dimostrata, potrebbe essere un'ottima scelta).

Si è visto come gli ambiti di utilizzo di tecnologie di Stream Processing siano molto vari: manutenzione, controllo di flussi di lavoro, predizioni, statistiche e molto altro. Il lavoro di questa tesi è incentrato principalmente su due ambiti: analytics e anomaly detection.

È stata realizzata una soluzione per il processamento di dati provenienti dai caselli autostradali, che corrispondevano a transazioni di ingresso o uscita. Queste informazioni sono state iniettate nella piattaforma Apache Kafka, con le varie librerie e componenti aggiuntivi, tra i quali Kafka Streams e Kafka Connect. L'applicazione effettua la collezione dei dati tramite Topic Kafka, opportunamente configurati; successivamente, è stata creata una topologia che processa i dati, li aggrega tramite OBU, effettua l'estrapolazione dei dati binari relativi alle tratte autostradali per poi suddividerli in finestre temporali. Quest'ultime sono i quarti d'ora, le ore e i giorni, utilizzate per poter effettuare query temporali a posteriori, con granularità di un quarto d'ora. Oltre questo, è stata aggiunta una parte di topologia che filtrasse eventuali dati anomali, ovvero transazioni fallite consecutivamente superiore ad un certo threshold, che sono poi iniettate in un opportuno Topic, da poter monitorare con consumatori appositi. La parte finale dell'applicazione è il modulo per le REST API, che espone la possibilità di effettuare ogni tipo di query possibile, con relativo filtraggio per OBU e per EID.

Non sono state rilevate problematiche particolari, a parte l'impossibilità di poter effettuare test sull'ambiente di rilascio ufficiale, cosa che rende, purtroppo, i test di performance non troppo affidabili e completi.

Possibili sviluppi futuri, che non sono stati implementati in questo progetto in quanto non ritenuti utili dall'azienda, sono di progettare un sistema di estrapolazioni predittive, magari per predire eventuali fault degli OBU, oppure un possibile traffico in una certa parte dell'autostrada a fronte di un gran numero di transazioni da un singolo casello.

# Sitografia

[1] Big Data 4 Innovation, https://www.bigdata4innovation.it/

[2] Stream Processing, Hazelcast, https://hazelcast.com/glossary/stream-processing/

[3] Piattaforme di Stream Processing a confronto, Medium, https://medium.com/@chandanbaranwal/spark-streaming-vs-flink-vs-storm-vs-kafka-streams-vs-samza-choose-your-stream-processing-91ea3f04675b

[4] Corso di Sistemi Distribuiti, Unibo, Paolo Bellavista, http://lia.deis.unibo.it/Courses/sd2021-info/

[5] Engineering, https://www.engineering.it/

[6] Documentazione ufficiale Apache Kafka, https://kafka.apache.org/documentation/

[7] Documentazione Confluent Kafka Connect, https://docs.confluent.io/platform/current/connect/index.html

[8] Documentazione ufficiale Avro, https://avro.apache.org/docs/1.10.2/

[9] Documentazione Confluent Kafka Registry, https://docs.confluent.io/platform/current/schema-registry/index.html

[10] Documentazione Confluent Kafka Streams, https://www.confluent.io/blog/introducing-kafka-streams-stream-processing-made-simple/

[10] Documentazione ufficiale Spring, https://docs.spring.io/spring-framework/docs/current/reference/html/

[11] Documentazione ufficiale Spring Boot, https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/